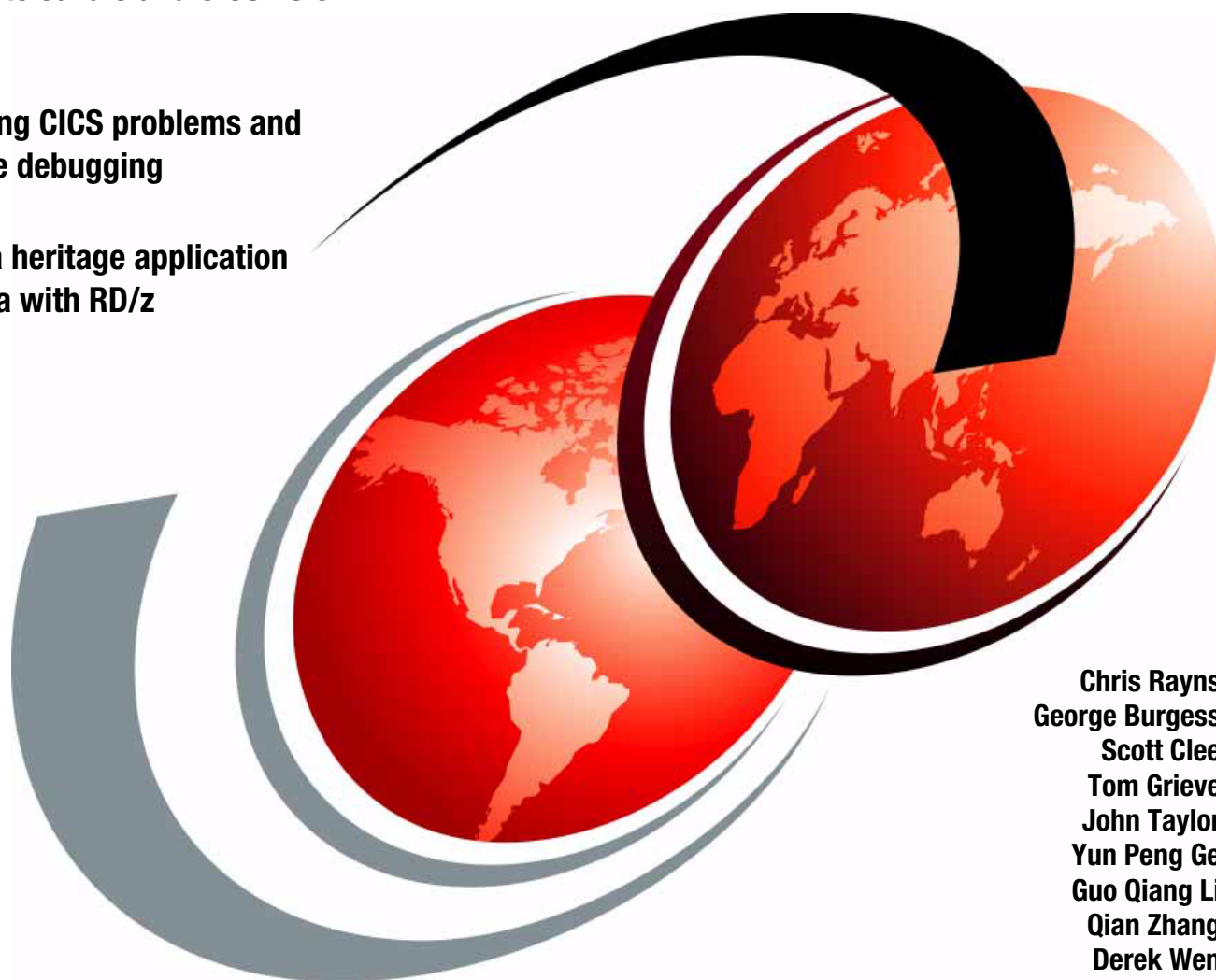


Java Application Development for CICS

Migrating to Java 5 and CICS TS 3.2

Determining CICS problems and interactive debugging

Evolving a heritage application using Java with RD/z



Chris Rayns
George Burgess
Scott Clee
Tom Grieve
John Taylor
Yun Peng Ge
Guo Qiang Li
Qian Zhang
Derek Wen

Redbooks



International Technical Support Organization

Java Application Development for CICS

February 2009

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

Fourth Edition (February 2009)

This edition applies to Version 3, Release 2, CICS Transaction Server .

© Copyright International Business Machines Corporation 2009. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this book	xi
Become a published author	xiii
Comments welcome	xiii
Summary of changes	xv
February 2009, Fourth Edition	xv
Part 1. Overview	1
Chapter 1. Introduction	3
1.1 z/OS	4
1.2 CICS Transaction Server Version 3	6
1.3 Java overview	7
1.3.1 Java language	7
1.3.2 Java Virtual Machine	9
1.3.3 Java on z/OS	10
1.3.4 Runtime Environment and tools	10
1.4 CICS Transaction Server for z/OS 3.2 enhancements for Java	13
1.4.1 Usability enhancements	13
1.4.2 Java Virtual Machines management enhancements	13
1.4.3 Continuous Java Virtual Machines versus resettable Java Virtual Machines	14
1.4.4 CICS Java applications using JCICS	14
1.4.5 CICS support for the Java Virtual Machine	14
Chapter 2. Java Virtual Machine support in CICS	17
2.1 Overview	18
2.2 History of JVM support in CICS	18
2.2.1 CICS Transaction Server 1.3	18
2.2.2 CICS Transaction Server 2.1 and 2.2	19
2.2.3 CICS Transaction Server Version 2.3	19
2.2.4 CICS Transaction Server 3.1	19
2.2.5 CICS Transaction Server 3.2	19
2.3 JVM operation modes	20
2.3.1 Single use JVM	20
2.3.2 Continuous JVM	21
2.3.3 Resettable JVM	23
2.3.4 Summary of JVM modes	23
2.4 Analyzing programs for use in a continuous JVM	23
2.4.1 Configuring the application isolation utility on UNIX System Services	24
2.4.2 Generating reports on static updates	25
2.5 The shared class cache	27
2.5.1 Benefits of the shared class cache	28
2.5.2 Java 5 shared class cache	28
2.5.3 Java 1.4.2 shared class cache	28
2.5.4 Starting the shared class cache	29

2.5.5 Inquiring the status of the shared class cache	29
2.5.6 Changing the size of the shared class cache	32
2.5.7 Updating classes in the shared class cache	32
2.5.8 The -Xshareclasses utilities	33
Part 2. Systems Programming	35
Chapter 3. Setting up CICS to run Java applications	37
3.1 Running a simple Java application in CICS	38
3.1.1 Accessing the z/OS UNIX shell	38
3.1.2 Setting up the CICS sample Java application	40
3.2 System configuration	44
3.2.1 UNIX System Services	44
3.2.2 Language Environment	45
3.2.3 CICS Transaction Server	46
3.3 Managing your CICS Java environment	53
3.3.1 CEMT INQUIRE CLASSCACHE	53
3.3.2 CEMT INQUIRE DISPATCHER	54
3.3.3 CEMT INQUIRE JVM	54
3.3.4 CEMT INQUIRE JVMPOOL	55
3.3.5 CEMT INQUIRE PROGRAM	55
3.3.6 CEMT PERFORM CLASSCACHE	56
3.3.7 CEMT PERFORM JVMPOOL	57
3.3.8 CEMT SET DISPATCHER	58
3.3.9 CEMT SET JVMPOOL	58
Part 3. Java programming for CICS	61
Chapter 4. Getting started	63
4.1 Coding your application in Rational Developer for System z	64
4.2 Deploying and running the program	69
4.2.1 Deploying the code to CICS	69
4.2.2 Setting up the transaction and program definitions	71
4.2.3 Running the program	72
4.2.4 Troubleshooting	74
Chapter 5. Writing Java 5 applications for CICS	75
5.1 Migrating Java applications to Java 5	76
5.2 New compiler errors and warnings	76
5.2.1 Error: syntax error on token 'enum'	76
5.2.2 Warning: ClassX is a raw type. References to generic type ClassX<E> should be parameterized	76
5.2.3 Removed Error for boxing/unboxing	77
5.3 Using the new features in Java 5	77
5.3.1 Generics	77
5.3.2 Enhanced for loop	79
5.3.3 Autoboxing and unboxing	80
5.3.4 Typesafe enums	81
5.4 Introduction to CICS for Java programmers	83
5.5 CICS program design guidelines	84
5.6 Differences in Java with CICS	85
5.6.1 Threads	85
5.6.2 Sockets	86
5.6.3 File I/O	86

5.6.4	Static data	86
5.6.5	Modifying the JVM state	87
5.6.6	Releasing resources at the end of program execution	87
5.6.7	Object Request Broker (ORB)	87
5.7	Data type conversion	87
5.7.1	ASCII & EBCDIC issues	87
5.7.2	Conversion to and from COBOL, PL/I, and Assembler data types	88
Chapter 6.	The Java CICS API	89
6.1	Introduction to JCICS	90
6.2	A short overview of the JCICS API	91
6.2.1	Program control	91
6.2.2	File control	92
6.2.3	Synchronization	92
6.2.4	Scheduling services	92
6.2.5	Unit of work	92
6.2.6	Document services	92
6.2.7	Web and TCP/IP services	93
6.2.8	Transient storage queues	93
6.2.9	Transient data queues	93
6.2.10	Terminal control	93
6.2.11	Miscellaneous services	94
6.2.12	Services that the JCICS API does not support	95
6.3	JCICS basics	95
6.4	Input and output streams	96
6.5	Exception handling	96
6.6	Calling other programs and passing data	101
6.6.1	Calling other programs using LINK and XCTL	103
6.6.2	Passing data between programs	103
6.6.3	Communicating using the COMMAREA	104
6.6.4	Communicating through Channels and Containers	105
6.6.5	COMMAREAs versus channels and containers	107
6.7	Remoteable resources	108
6.8	Using transient storage queues	109
6.9	Performing serialization	112
6.10	Web, TCP/IP, and document services	114
6.11	File control	115
6.12	Interval control	124
6.13	Terminal services	125
6.14	Using JZOS with CICS	134
Chapter 7.	Evolving a heritage application using Java	135
7.1	The heritage Trader application	136
7.1.1	Installing the Trader application	136
7.2	Other Extensions to the Trader application	137
7.2.1	Using WMQ Classes to drive the Trader application	137
7.2.2	Using the CICS Common Client Interface (CCI)	138
7.3	Adding a JCICS Web interface	138
7.3.1	Wrapping the COMMAREA	139
7.3.2	Wrapping the COMMAREA using JZOS	142
7.3.3	Wrapping the COMMAREA using J2C in RD/z	144
7.3.4	Understanding COMMAREA request formats	148
7.3.5	A test Web application	148

7.3.6	Designing the HTML interface.	151
7.3.7	Implementing the design.	152
7.3.8	Setting up TraderPL	157
7.3.9	Web security	160
7.4	Migrating TRADERBL to JCICS	161
7.4.1	Mapping COBOL to Java	162
7.4.2	Using TraderBL with VSAM	169
7.4.3	Setting up TraderBL	170
7.5	Moving to a DB2 back end	171
7.5.1	Data migration.	172
7.5.2	Changing the JVM profile for DB2.	173
7.5.3	Using TraderBL with DB2	174
7.5.4	Setting up TraderBL with DB2	177
7.6	Adding a Web services interface.	178
7.6.1	Building the TraderBL Web service provider.	178
7.6.2	Migrating TraderPJ to a Web service requester	185
Chapter 8.	Problem determination and debugging	193
8.1	Debugging and problem determination.	194
8.1.1	First considerations.	194
8.2	Common problems	196
8.2.1	Abend AJ04	196
8.2.2	Incorrect output or behavior	196
8.2.3	No response	197
8.2.4	OutOfMemoryError	197
8.2.5	Performance is not good.	198
8.2.6	Problems caused by static values in continuous JVMs.	198
8.3	Where to look for diagnostic information.	199
8.3.1	Javadumps	199
8.3.2	Heapdump	203
8.3.3	Monitoring garbage collection cycles	204
8.3.4	JVM stdout and stderr.	206
8.3.5	JVM method tracing	207
8.3.6	JVM class loader tracing.	209
8.3.7	Shared classes diagnostics	211
8.4	Interactive debugging	212
8.4.1	Execution diagnostic facility	212
8.4.2	Debugging using Rational Developer for System z.	213
8.4.3	CICS Application Debugging Profile	217
8.4.4	The CICS JVM plug-in mechanism.	218
Chapter 9.	Performance for Java in CICS Transaction Server Version 3	221
9.1	Reusable Java virtual machine	222
9.1.1	CICS Task Control Blocks and the Java virtual machine	222
9.1.2	The reusable Java virtual machine	222
9.1.3	Removing resettable mode for JVMs in CICS Transaction Server 3.2	223
9.2	Shared Class Cache facility	223
9.2.1	Overview of the Shared Class Cache facility	224
9.3	Things to avoid	225
9.3.1	Java virtual machine stealing	225
9.3.2	Using application classpath	226
9.3.3	Excessive garbage collection	227
9.4	IBM zSeries Application Assist Processor specialty engines	227

9.4.1	zAAP introduction	227
9.4.2	zAAP benefits	228
9.4.3	zAAP requirements	228
9.4.4	zAAP workflow	229
9.4.5	Using zAAPs in JVM	230
 Chapter 10. Performance tools for Java in CICS Transaction Server		
	Version 3	233
10.1	CICS Explorer	234
10.1.1	System requirements	234
10.2	CICS PA overview	237
10.3	CICSplex System Management	245
10.4	OMEGAMON XE for CICS on z/OS	257
 Part 4. Appendix		
Appendix A. JCICS exception mapping		291
Appendix B. Hints and tips		293
Priority of public static void main() methods		294
Getting transaction arguments using Java		294
Never use System.exit()		295
 Appendix C. Resettable JVM		297
Resettable JVM		298
 Related publications		301
IBM Redbooks		301
Other publications		301
Online resources		301
How to get Redbooks		302
Help from IBM		302
 Index		303

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.


COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

1-2-3®	IMS™	Redbooks (logo)  ®
AFS®	Language Environment®	S/390®
AIX®	Lotus®	SupportPac™
alphaWorks®	MVS™	System z9®
CICSplex®	NetView®	System z®
CICS®	OMEGAMON II®	Tivoli®
Common User Access®	OMEGAMON®	VisualAge®
CUA®	OS/2®	VTAM®
DB2®	OS/390®	WebSphere®
developerWorks®	OS/400®	z/Architecture®
DFS™	Parallel Sysplex®	z/OS®
e-business on demand®	PR/SM™	z/VM®
ETE™	RACF®	z10™
HiperSockets™	Rational®	z9®
IBM®	Redbooks®	zSeries®

The following terms are trademarks of other companies:

EJB, Enterprise JavaBeans, J2EE, Java, JavaBeans, Javadoc, JDBC, JDK, JNI, JRE, JVM, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

We wrote this IBM® Redbooks publication for clients who implement the Java™ language support that is provided by CICS® Transaction Server for z/OS® V3.2. Our prime audience is CICS and z/OS system programmers who provide support for Java application development and Java application programmers who need a gentle introduction to Java development for CICS.

In this book, we assume that you have knowledge of z/OS, CICS, UNIX® System Services, and Java.

We start by reviewing the basic concepts of the z/OS, CICS TS V3.2, and Java environments, and introduce new terminology. We then discuss the software and hardware requirements for developing and executing Java applications in CICS TS V3.2. Next we show you how to customize the application development environment, UNIX System Services, MVS™, and CICS.

Additionally, we briefly discuss three possible application development roadmaps: Java application programs that use CICS services, IIOF server applications, and CICS Enterprise Beans.

Subsequent chapters contain an expanded explanation and examples of Java application programs that use CICS services and how to use CICS-supplied Java class library and the Java Virtual Machine (JVM™). We then present a CICS business application that has presentation and business logic.

Finally, we provide guidance on debugging and problem determination.

The team that wrote this book

This book was produced by a team of specialists from around the world working at Beijing, China.

Chris Rayns is an IT Specialist and Project Leader at the ITSO, Poughkeepsie Center in New York, where he specializes in security. Chris writes extensively on all areas of IBM S/390® security. Before joining the ITSO, he worked in IBM Global Services in the United Kingdom (UK) as a CICS IT Specialist.

George Burgess is currently working as the CICS Transaction Server on z/OS Subject Matter Expert for the Peoples Republic of China and is based in Beijing. He has 24 years of experience as an Application Programmer, Systems Programmer, CICS Developer and OMEGAMON® XE for CICS Developer. His areas of expertise include Common Business Oriented Language (COBOL), CICS, DB2®, WebSphere® MQ, IMS™ DL/1, VSAM, JCL, z/OS, and OMEGAMON.

Scott Clee is the CICS Test Architect for IBM in the United Kingdom (UK). He frantically architects the face of Test by looking for new ways to push Testing techniques, process, and technology. He has a degree in Mathematics and Computing from the University of Bath, UK. His areas of expertise include CICS, Java, Common Business Oriented Language (COBOL), Linux®, and Testing. Check out his blog at TestingBlues.com.

Tom Grieve Tom Grieve is a Software Engineer in CICS Development at Hursley Park, UK. He has a Bachelor of Science degree in Mathematics and Physics from the University of London and is currently studying for a Masters in Software Engineering at the University of Oxford. He has 35 years of experience as an Application Programmer, Systems Programmer, and Software Engineer. He has worked at IBM for the last five years in CICS and previously as a Developer on CICS for OS/2® and the Java Virtual Machine for OS/390®. His areas of expertise include CICS and Java.

John Taylor is a Software Engineer in CICS Development at Hursley Park, UK. He has a Bachelors degree in Digital Systems Engineering from the University of the West of England, Bristol and a Masters degree in Software Engineering from the University of Oxford. He has 12 years of experience as an Application Programmer, Tester, CICS Developer, and OMEGAMON® XE for CICS Developer with IBM. His areas of expertise include CICS, Java, OMEGAMON, and various forms of tooling.

Yun Peng Ge is an Advisory IT Specialist in the Technical Sales Support team supporting mainframe WebSphere customers in China. He has a Bachelors degree in Computer Science from Fudan University in Shanghai. His areas of expertise includes CICS Transaction Server, WebSphere MQ, WebSphere Message Broker, J2EE, and z/OS.

Guo Qiang Li is a Software Engineer with the China CICS Team, which is the first team in CDL working on CICS Transaction Server. He graduated from Tianjin University with a Masters degree and joined the China CICS Team in 2006. He focuses on CICS Dynamic LIBRARY management testing and CICS Web Service support testing. His experiences on CICS include CPSM and Web Services.

Qian Zhang is an Advisory IT Specialist with the IBM ATS team of China, since 2004. He has eight years of experience working for IT development and IT support. His expertise resides in mainframe system knowledge, which includes CICS TS, Java, z/OS, MVS, and parallel Sysplex. He has domestic and international experience in proof of concept and performance benchmarking on System z®. Between 2005 and 2007, he worked as a System Programmer at the IBM China System Center. He is currently committed to supporting CICS TS performance projects in China.

Derek Wen is an IT Specialist in IBM Global Technology Services in Taiwan. He has over 20 years of experience in the IBM mainframe field and over 15 years of experience with CICS. His areas of expertise include zOS, MVS, CICS, Assembler, COBOL, and Java with a focus on heritage application integration with CICS.

Thanks to the following people for their contributions to this project:

Richard M Conway
International Technical Support Organization, Poughkeepsie Center

Paul Cooper, CICS TS for z/OS Development
IBM Hursley

Joe Winchester, Software Developer
IBM Hursley

Dennis Weiland, ATS
IBM Dallas

Thanks to the authors of the previous editions of this book.

- Authors of the first edition, *Java Application Development for CICS*, published in August 2005, were:

Scott Clee

Ulrich Gehlert

Vasilis Karras

Debra Payne

Bill Plowman

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks® in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- Send your comments in an e-mail to:

redbooks@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Summary of changes

In this section, we describe the technical changes that we made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-5275-03
for Java Application Development for CICS
as created or updated on February 24, 2009.

February 2009, Fourth Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- ▶ Performance tools for Java in CICS TS V3
- ▶ CICS Web Services support in the Trader application
- ▶ CICS TS Explorer
- ▶ CICS Performance Analyzer
- ▶ CICSplex WUI
- ▶ Rational Developer for z

Changed information

- ▶ All chapters changed to reflect CICS TS 3.2 changes



Part 1

Overview

In part one of this book, we familiarize you with the basic concepts of and developments in object-oriented (OO) technology in relation to the z/OS and CICS Transaction Server for z/OS (CICS TS). Java has rapidly grown in popularity throughout the Information Technology (IT) industry and is now the programming language of choice for many organizations.



Introduction

In this chapter, we familiarize you with the basic concepts of and developments in object-oriented (OO) technology in relation to the z/OS and CICS Transaction Server for z/OS (CICS TS). Java rapidly grew in popularity throughout the Information Technology (IT) industry and is now the programming language of choice for many organizations. CICS extended the support for Java-based workloads over a number of releases in response to the uptake of the Java programming model. CICS Transaction Server for z/OS Version 3 adds further support for Java, which includes the provision of greater controls for the Runtime Environment that Java application programs use.

The main topics that we cover in this chapter are:

- ▶ Positioning of z/OS, CICS TS, and Java
- ▶ Using Java in a CICS environment

1.1 z/OS

z/OS is a highly secure, scalable, high-performance enterprise operating system on which you can build and deploy Internet and Java-enabled applications, providing a comprehensive and diverse application execution environment.

z/OS:

- ▶ Provides a highly secure, scalable, high-performance base for on demand applications
- ▶ Can simplify IT infrastructure by allowing the integration of applications in a single z/OS image
- ▶ Takes advantage of the latest open software technologies to extend existing applications and to add new on demand applications
- ▶ Incorporates world-class optimization features, security services, distributed print services, storage management, and Parallel Sysplex® availability

z/OS is the robust IBM eServer zSeries® mainframe operating system that is designed to meet the demanding quality of service requirements for on demand business. With the IBM eServer zSeries servers, z/OS serves as the heart your on demand infrastructure.

z/OS takes advantage of the latest open and industry software technologies, such as Enterprise JavaBeans™, XML, HTML, C/C++, and Unicode. z/OS UNIX System Services allows you to develop and run UNIX programs on z/OS and exploit the reliability and scalability of the z/OS platform.

z/OS also incorporates world-class optimization features, security and IP networking services, distributed print services, storage management, and Parallel Sysplex availability.

Some of the unique classic strengths of z/OS are:

On demand infrastructure

z/OS has optimization features to help provide the responsiveness needed for on demand applications. The z/OS Workload Manager (WLM) is at the heart of z/OS optimization and is designed to manage the priority of mixed workloads based on business policies that are defined in Service Level Agreement terms. Intelligent Resource Director (IRD) extends the z/OS Workload Manager to work with PR/SM™ on zSeries servers with features to dynamically manage resources across multiple logical partitions (LPARs). Based on business goals, WLM is designed to adjust processor capacity, channel paths, and I/O requests in real time across LPARs without human intervention.

Together with the IBM zSeries servers and with interoperability with Linux for zSeries and z/VM®, z/OS can play a critical role in simplifying your infrastructure. Both z/OS and Linux for zSeries support much of the server differentiation that sets the zSeries apart from other servers. With zSeries servers, z/OS provides the base for the z/Architecture® with support for 64-bit storage, Intelligent Resource Director (IRD), and HiperSockets™ (for inter-partition communications).

Performance

High-volume transaction processing and heavy batch processes are common on z/OS. z/OS is efficient in managing its use of hardware and software resources.

Availability	The high availability of z/OS is a key reason why so many clients rely on z/OS for their most critical applications. To continue to improve this availability, z/OS provides automation capabilities in a new element, Managed System Infrastructure for Operations. This element provides automation for single system and sysplex operations to help simplify operations and improve availability.
Self-configuring	z/OS Managed System Infrastructure for Setup (msys for Setup) is the z/OS solution for simplifying product installation, configuration, and function enablement. msys for Setup uses wizard-like configuration dialogs, which helps to reduce configuration errors. msys for Setup provides multi-user and multi-system support. Also, msys for Setup can use the IBM Directory Server, OpenLDAP, on any IBM platform, which includes OpenLDAP on z/OS, which can simplify the initialization of msys for Setup.
Scalability	z/OS is a highly scalable operating system that can support the integration of new applications. z/OS can scale up in a single logical partition and scale out in a Parallel Sysplex cluster for higher availability.
64-bit support	The z/OS scale is extended with support for 64-bit real and virtual storage on System z servers, while continuing to support 24-bit and 31-bit applications. The 64-bit real support eliminates expanded storage, helps eliminate paging, and might allow you to consolidate your current systems into fewer LPARs or to a single native image. z/OS delivers 64-bit virtual storage management support. This 64-bit support is used by DB2 V8 and other middleware.
Security	z/OS extends its robust mainframe security features to address the demands of on demand enterprises. Technologies, such as Secure Sockets Layer (SSL), Kerberos, Public Key Infrastructure, multilevel security, and exploitation of System z cryptographic features, are available in z/OS. Integrated Cryptographic Service Facility (ICSF) is a part of z/OS, which provides cryptographic functions for data security, data integrity, personal identification, digital signatures, and the management of cryptographic keys. Together with cryptography features of the IBM System z servers, z/OS provides high-performance SSL, which can benefit applications that use z/OS HTTP Server and WebSphere, TN3270, and CICS Transaction Gateway server.

z/OS provides support for digital certificates, including the ability to provide full life cycle management. With Public Key Services in z/OS, you can create and manage digital certificates and leverage your existing z/OS mainframe investments, which can provide significant cost savings over other digital certificate hosting options.

TCP/IP networking

z/OS provides TCP/IP network support with high performance, security, and scale. High availability is provided across a Parallel Sysplex where TCP/IP traffic uses the Sysplex Distributor and Dynamic VIPA (virtual IP addressing). Together with the Workload Manager, these functions are designed to provide failure independence in the face of TCP/IP or system outages. Load balancing of network traffic can be self-optimized with z/OS, which enables consistent response times for critical traffic in a complex multi-application network.

z/OS evolved to a fully open operating system that supports:

- ▶ Internet Web serving
- ▶ High security and integrity required for electronic commerce
- ▶ Distributed object-oriented application architectures

This support, combined with the classic strengths of z/OS, makes it an excellent operating system for your current and future mission-critical applications and data.

1.2 CICS Transaction Server Version 3

CICS Transaction Server for z/OS Version 3 provides an efficient and effective environment for applications that are written in COBOL, PL/I, C, C++, and Java. This version strengthens application development capabilities, enables enhanced re-use of 3270 applications, and enables applications to be accessed as Web Services within a services-oriented architecture (SOA).

The transaction processing strengths of CICS in an enterprise computing environment are appreciated and exploited worldwide. CICS has always provided a reliable transaction processing environment that:

- ▶ Provides a robust, high-performance runtime environment for enterprise applications written in Java.
- ▶ Supports EJB™ session beans, providing another dimension for application architects. Where an EJB component needs to incorporate procedural logic modules to accomplish its business function, CICS enables this mixed-language component to run in a single execution environment with good isolation from other components, improving robustness and manageability.
- ▶ Provides a runtime environment that is optimized for business logic written as EJBs that can run alongside, and interoperate with, business logic that is written in languages, such as COBOL. Both EJB applications and COBOL applications can access existing (and new) DB2, IMS DB, and VSAM data concurrently and with complete integrity.
- ▶ Provides enhancements for applications that use TCP/IP communication for e-business enablement. These offer a range of benefits in terms of management and improved scalability and performance.

- ▶ Provides enhanced DB2 facilities, which provides a significant improvement in performance and a greater level of availability.
- ▶ Assists the evolution to on demand computing through integration, openness, autonomic computing and virtualization.

1.3 Java overview

Java has become a popular programming language and runtime environment for building new applications on all platforms, including the System z and System z mainframes. Of course, in many mainframe installations new programs are created in languages, such as COBOL, C/C++, or PL/I, but with the increasing use of middleware, such as WebSphere Application Server, the Java programming model is also expanding. However, Java is not only used in relatively new middleware, such as WebSphere Application Server, but also in the traditional transaction managers, such as CICS, IMS, and DB2. Using Java in CICS is the theme of this book. In a standalone environment, you can use Java for programs that are submitted from a UNIX System Services command line or in JCL. This form of Java is not running inside middleware, a transaction, or a database server.

The Java specifications are maintained by Sun™ Microsystems, but other companies, IBM in particular, provide input for these specifications. The Java language is based on a philosophy:

Develop once, run everywhere

Java was originally developed in the early 90s by Sun Microsystems Inc. as an object-oriented programming language. The syntax used in Java code originates from C++ and is therefore recognizable for those with C++ experience. However, there are big differences between C++ and Java:

- ▶ The Java programming model was expanded with numerous APIs that are organized in libraries. The Java 2 Enterprise Edition (J2EE™) programming model goes even further and defines a syntax for writing programs and a way of packaging applications and much more.
- ▶ Java programs are not compiled into persistent load modules; however, technologies were added over the years to do compiles “on the fly” while running the applications. The first technology to support this was the Just-In-Time compiler (JIT). Later on, more technologies were included in the Java Virtual Machine (JVM) to speed up this process.
- ▶ Java, or actually the JVM, has its own memory management. The programmer does not need to worry about memory allocation and de-allocation because the JVM does this.

Over the years, Java grew to be popular, and the JVM is available on nearly all available operating systems, such as Microsoft® Windows®, Apple OS-X, OS/400®, Linux, UNIX, and z/OS UNIX.

1.3.1 Java language

The Java programming language is unusual because Java programs are both compiled (translated into an intermediate language called Java bytecodes) and interpreted (bytecodes parsed and run by the JVM). Compilation occurs once, although interpretation happens each time the program is run. Compiled bytecode is a form of optimized machine code for the JVM. The interpreter is an implementation of the JVM.

Java is object oriented but without all of the complications. It has a single inheritance model, simple data types, and code that is organized into classes. These classes provide an excellent way of packaging functions.

As a language, Java is statically typed and most types of checking occur at compile time. However, runtime checks, such as array bounds, are still required. A key function of Java is that numeric precision is defined with the IEEE floating point notation, which ensures the same results everywhere for mathematical calculations.

The sample Java source code in Example 1-1 on page 8 shows some Java source code (the typical "Hello World" program) and the basic structure and elements of a Java application. It begins with block documentation comments that are started with `/**` and end with `*/`. If this program were processed by the Java-Javadoc utility, a standard part of the Software Development Kit (SDK), these comments along with the structure of the program, its methods, and other documentation comments, would be automatically included in a documentation file in HTML format.

A second form of comment can also be seen in the program: `// display string`. This is a private comment and is not included in the HTML file that Javadoc™ produces.

Java also supports a third style of comment: The classical C comment that starts with `/*` and ends with `*/`. These are also private and are not included in Javadoc HTML files.

Example 1-1 shows sample Java source code.

Example 1-1 Sample Java source code

```
/**
 * HelloWorldApp class implements an application that simply
 * displays "Hello World" to the standard output device.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //display string
    }
}
```

In Example 1-1, after the documentation comment, the name of the class is defined: HelloWorldApp. This part of the program gives us important information, such as whether this is an application, an applet, or a servlet. In this case, the sample program is an application (it has a `main()` construct in its source code), and you can execute it from a command line.

The name of the class is case sensitive and must match the name of the file. The file type is `.java`, which makes the full file name `HelloWorldApp.java`. The file name (for example, `javac HelloWorldApp.java`) is used when you compile the program. A Java program is compiled into a `.class` file. The class name without its file type of `.class` (for example, `java HelloWorldApp`) is used when you run the program.

The next part of the program defines the properties of the Java program or class. In this case, it is a public class that can be called by any other class. Our sample program returns no data, and it takes as input a character string of command line arguments.

Finally, our sample program performs its function: In this case, to print out "Hello World" by creating an instance of a Java class. If you are only familiar with procedural programming, you can think of this as a call to a system-provided function. In reality, it is different, but that is beyond the scope of this introduction.

The source code shown in Example 1-1 is compiled into machine-independent bytecode with the Java compiler, `javac`. The bytecode is conceptually similar to an object deck in z/OS terms. It is executable but still needs to be link-edited.

Java bytecode is dynamically linked, which means that functions that are external to the program are located and loaded at runtime rather than being statically bound to the Java bytecode. Thus, functions can be loaded on demand over the network when needed, with unused code never loaded. If the Java program is an application or servlet, it can still be dynamically linked by loading classes over a network-based file system, such as network file system (NFS), distributed file system (DFS™), or Andrew file system (AFS®). Typically, however, all of the required classes are installed locally for applications and servlets.

After it is loaded and linked, Java bytecodes are ready for execution. Originally Java bytecode was always interpreted (translated) into native instructions. However, by default almost all JVMs include a just-in-time (JIT) compiler. The JIT compiler dynamically generates machine code for frequently used bytecode sequences in Java applications and applets while they are running.

The IBM Software Developer Kit for z/OS (SDK), Java 2 Technology Edition, which became generally available in September, 2002, provides the SUN SDK1.4 APIs and is periodically updated with cumulative service and improvements. The SDK for z/OS includes the JIT, which is enabled by default. You can disable the JIT to help isolate a problem with a Java application, an applet, or the compiler itself.

Because Java bytecode is in a machine-independent, architecture-neutral format, it can be run on any system with a standard Java implementation. An extensive library of underlying classes or functions can be used to do everything from graphics to network communication. Because these classes are Java bytecode, they too are machine independent.

1.3.2 Java Virtual Machine

At the core of the Java concept and implementation is the JVM, a complete software microprocessor with its own instruction set and operation (op)-codes. The JVM provides automatic memory management, garbage collection, and other functions for the programmer.

The IBM JVM, and most other JVMs, are implemented through licensed source code from Sun Microsystems. The source code is provided in C and Java and is highly portable. IBM ported it to many platforms: IBM AIX®, OS/2, OS/400, z/OS, and others.

The JVM uses z/OS UNIX System Services for z/OS for base operating system functions; therefore, you must correctly install and tune UNIX System Services to get optimum performance from the JVM.

The JVM is the "essence" of Java because it provides the machine independence that is the most significant advantage of Java. Although the JVM is not a unique concept and there were other software microprocessors over the past 20 years, it is the first and only one to achieve broad acceptance. This acceptance is primarily a result of Sun Microsystems making the source code for the JVM available under license. It is much quicker to implement through the source code than from scratch working from a reference document.

For more information about the Java Virtual Machine see Chapter 2, "Java Virtual Machine support in CICS" on page 17.

1.3.3 Java on z/OS

IBM is a major supporter and user of Java across all of the IBM computing platforms, which includes z/OS. The z/OS Java implementation provides the same full function Java APIs that are on all other IBM platforms. Additionally, the z/OS Java program products were enhanced to allow Java access to z/OS-unique file systems. Also, Java on z/OS provides a z/OS implementation of the Java Native Interface (JNI™).

The Java implementation on z/OS, as on other platforms, includes an Application Programming Interface (API) and a Java Virtual Machine (JVM) to run programs. The existence of a Java Virtual Machine means that applications written in Java are largely independent of the operating system used.

Note: To stimulate the use of Java on the mainframe, IBM introduced a new specialty processor for running Java applications called the System z Application Assist Processor, also known as zAAP. This type of processor is an optional feature in the System z9® and z10 hardware. After installed and enabled, it allows you to benefit from additional resources that are available for Java code, and in some select cases, non-Java code closely related to the execution of Java. For zAAP processors, no software license fees are paid for certain IBM software products. Using zAAP processors, you can expand the system's CPU capacity at a relatively low cost, if the workload that is run is based on Java.

On the hardware level settings, in PR/SM, zAAPs are treated and managed as a separate pool of logical processors. So the weight factors can be different from what you have in place for the General Purpose (GP) processors.

For zAAP processors, special hardware and operating system requirements exist. For more information about the zAAP processors, see *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177.

Java Software Development Kit on z/OS

Java Software Development Kits (SDKs) contain application programming interfaces (APIs). You can order and service each SDK product independently and separately. The Java SDKs for z/OS are available electronically through the Internet or by placing a regular software order at IBM Software Manufacturing. Visit the following Web site for more information about these products and for download instructions:

<http://www-03.ibm.com/servers/eserver/zseries/software/java/allproducts.html>

At this time the following SDK products for z/OS are available:

- ▶ IBM SDK for z/OS, Java 2 Technology Edition, V1.4 (5655-I56), SDK1.4.2
- ▶ IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V1.4 (5655-M30), SDK1.4.2
- ▶ IBM 31-bit SDK for z/OS, Java 2 Technology Edition, V5 (5655-N98), SDK5
- ▶ IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V5 (5655-N99), SDK5

All of these SDK products are available in a non-SMP/e installable format and an SMP/e installable flavor.

As time and technology progress, products are withdrawn from service and new ones are introduced.

1.3.4 Runtime Environment and tools

The SDK contains a Java Runtime Environment (JRE™) and several development tools. In this section, we describe the contents of the Runtime Environment and the SDK tools.

Runtime Environment

The Runtime Environment contains:

- ▶ Core classes: The compiled class files for the platform that must remain zipped for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.
- ▶ JRE tools: The following tools are part of the Runtime Environment and are in the `/usr/lpp/java/J5.0/bins` directory (where `/usr/lpp/java/J5.0/` is the directory in which you installed the SDK):
 - JVM runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.
 - Java Interpreter (`javaw`) runs Java classes in the same way that the `java` command does, but does not use a console window.
 - Key and Certificate Management Tool (`keytool`) manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.
 - Hardware Key and Certificate Management Tool (`hwkeytool`) works, such as `keytool`, but also allows you to generate key pairs and store them in a keystore file of type JCA4758KS.
 - Policy File Creation and Management Tool (`policytool`) creates and modifies the external policy configuration files that define your installation's Java security policy.
 - RMI activation system daemon (`rmid`) starts the activation system daemon so that objects are registered and activated in a Java virtual machine (JVM).
 - Common Object Request Broker Architecture (CORBA) Naming Service (`tnameserv`) that starts the CORBA transient naming service.
 - Java Remote Object Registry (`rmiregistry`) creates and starts a remote object registry on the specified port of the current host.
 - Dump extractor (`jextract`) converts a system-produced dump into a common format that `jdumpview` can use. For more information, see the relevant chapter in the IBM JVM Diagnostics Guide that is located at:

<http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html>

Earlier versions of the IBM JRE shipped with a file called `rt.jar` in the `jre/lib` directory. From Java v1.4 onwards, this file is replaced by multiple JAR files that reside in the `jre/lib` directory. Examples of these JAR files are:

- ▶ `core.jar`: contains the majority of the class libraries, which includes the system, IO, and net class libraries.
- ▶ `graphics.jar`: contains the awt and swing class libraries.
- ▶ `security.jar`: contains the security framework code. For maintenance reasons, `security.jar` is split up into smaller JAR files for this release.
- ▶ `v server.jar`: contains the RMI class libraries.
- ▶ `xml.jar`: contains the xml and html class libraries.

This change must be completely transparent to the application. If an error is received about a missing `rt.jar` file in `CLASSPATH`, this error points to a setting that was used in Java V1.1.8 and was made obsolete in subsequent versions of Java. You can safely remove references to `rt.jar` in `CLASSPATH`.

Software Development Kit tools

The following tools are part of the SDK and are located in the /usr/lpp/java/J5.0/bin directory:

- ▶ Java Compiler (javac) compiles programs that are written in the Java programming language into bytecodes (compiled Java code).
- ▶ Java Applet Viewer (appletviewer) tests and runs applets outside of a Web browser.
- ▶ Class File Disassembler (javap) disassembles compiled files and can print a representation of the bytecodes.
- ▶ Java Documentation Generator (javadoc) generates HTML pages of API documentation from Java source files.
- ▶ C Header and Stub File Generator (javah) enables you to associate native methods with code written in the Java programming language.
- ▶ Java Archive Tool (jar) combines multiple files into a single Java Archive (JAR) file.
- ▶ JAR Signing and Verification Tool (jarsigner) generates signatures for JAR files and verifies the signatures of signed JAR files.
- ▶ Native-To-ASCII Converter (native2ascii) converts a native encoding file to an ASCII file that contains characters that are encoded in either Latin-1, Unicode, or both.
- ▶ Java Remote Method Invocation (RMI) Stub Converter (rmic) generates stubs, skeletons, and ties for remote objects, which includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.
- ▶ IDL to Java Compiler (idlj) generates Java bindings from a given IDL file.
- ▶ Serial Version Command (serialver) returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.
- ▶ Extcheck utility (extcheck) detects version conflicts between a target jar file and currently installed extension jar files.
- ▶ Cross-platform dump formatter (jdmpview) is a dump analysis tool that allows you to analyze dumps. For more information, see the relevant chapter in the *IBM JVM Diagnostics Guide* located at:
<http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html>
- ▶ INCLUDE FILES: C headers for JNI programs.
- ▶ DEMOS: The demo directory contains a number of subdirectories that contain sample source code, demos, applications, and applets, that you can use.
- ▶ COPYRIGHT: Copyright notice for the SDK for z/OS software. The user guides and the accompanying copyright files and demo directory are the only documentation that are included in this SDK for z/OS. You can view Sun's software documentation by visiting the Sun Web site, or you can download Sun's software documentation package from the Sun Web site:
<http://java.sun.com>

The following tools are not included in the IBM SDK:

- ▶ MIF doclet
- ▶ orbd
- ▶ servertool

1.4 CICS Transaction Server for z/OS 3.2 enhancements for Java

CICS Transaction Server for z/OS, Version 3 Release 2 can support the JVM that is provided by the 31-bit version of IBM SDK for z/OS, Java 2 Technology Edition, Version 5.

1.4.1 Usability enhancements

JVM profile and properties changes:

- ▶ You can now specify any JVM option or system property
- ▶ You can specify system properties in JVM properties or profile

Improved error messages and trace output:

- ▶ Validation checks to address common user errors
- ▶ CICS formats JVM trace output

New Garbage collection in CICS TS 3.2

Garbage Collection scheduling algorithm in CICS changed:

- ▶ Occurs at a target heap utilization:
 - GC_HEAP_THRESHOLD parameter in the JVM profile (default 85%)
 - 100% implies no CICS-scheduled GC
- ▶ Performed asynchronously in a CICS system task:
 - The application does not suffer bad response times simply because a scheduled GC happened to take place

New transaction CJGC:

- ▶ Unscheduled GC within the JVM can still occur at any time:
 - But with a little tuning you can minimize the likelihood of this happening

1.4.2 Java Virtual Machines management enhancements

Specify a JVM timeout value:

- ▶ IDLE_TIMEOUT=30
- ▶ After timeout JVMs become eligible for termination

Pre-initialize JVMs by profile:

- ▶ PERFORM JVMPOOL START JVMCOUNT() JVMPROFILE()
- ▶ Performed asynchronously using CJPI transaction

Selectively phase out JVMs by profile:

- ▶ PERFORM JVMPOOL PHASEOUT JVMPROFILE()

1.4.3 Continuous Java Virtual Machines versus resettable Java Virtual Machines

Continuous JVMs perform better than resettable and more consistent with other versions of Java:

- ▶ Lower CPU cost per transaction
- ▶ Simpler to set up and tune (fewer different storage heaps)
- ▶ Compatible with future versions of Java

Resettable mode was deprecated in CICS TS 3.2:

- ▶ IBM JVM CICS Application Isolation Utility (support pac CH1B)

1.4.4 CICS Java applications using JCICS

You can write Java application programs that use CICS services and execute under CICS control.

You can write Java programs on a workstation or in the z/OS UNIX System Services shell. You can use any editor of your choice or a visual composition environment, such as Rational Application Developer.

CICS provides a Java class library, known as JCICS, supplied in the dfjcics.jar JAR file. JCICS is the Java equivalent of the EXEC CICS application programming interface that you use with other CICS supported languages, such as COBOL. It allows you to access CICS resources and integrate your Java programs with programs that are written in other languages. Most of the functions of the EXEC CICS API are supported. We discuss the JCICS API extensively in Chapter 6, “The Java CICS API” on page 89.

The Java language is designed to be portable and architecture-neutral. The bytecode that is generated by compilation is portable, but it requires a machine-specific interpreter for execution on different platforms. CICS provides this execution environment by means of a Java Virtual Machine that executes under CICS control.

1.4.5 CICS support for the Java Virtual Machine

Java has rapidly grown in popularity throughout the IT industry, and for many organizations, it is now their programming language of choice. CICS Transaction Server extended the support for Java technology-based workloads over a number of releases in response to the uptake of the Java programming model. CICS Transaction Server, Version 3.2 adds further support for Java, making this version essential for anyone who is already running Java workloads in earlier-version CICS environments.

From a historical perspective, the expectations that users had for CICS Transaction Server and Java matured over time. When IBM first introduced Java support in CICS Transaction Server, it was expected that Java might behave as similar to COBOL as possible from within the CICS runtime environment. Today, the expectation is that Java in CICS Transaction Server is much like Java on other platforms.

The unique characteristics of the persistent reusable Java Virtual Machine (JVM) that were traditionally supported in earlier versions of CICS Transaction Server make it more difficult to write Java programs that run as intended in CICS Transaction Server. As a result of this and other performance-related issues, many users of Java in a CICS Transaction Server, Version 2.3 environment elected to use the continuous JVM exclusively and abandoned the persistent reusable JVM altogether.

Recent versions of the JVM do not include the persistent reusable JVM extensions. To make CICS Transaction Server consistent with this Java change, CICS Transaction Server, Version 3.2 also does not support the persistent reusable JVM (support is for the continuous JVM only). Support for the class cache remains unaffected.

The persistent reusable JVM was first made available for use in CICS Transaction Server, Version 2.1. It offers the ability to reset the state of a JVM between tasks to help ensure that subsequent users of the same JVM are fully isolated from states left behind by previous users of the JVM. The time taken to reset a JVM depends on there being no cross-heap references between the middleware and the application heaps within the JVM. If these references exist, the JVM scans the heap to determine if the references are in live objects. The scan process is rather slow. If the attempt to reset the JVM fails, CICS Transaction Server discards the JVM and creates a new one. These unresettable events (UREs) are a major performance problem for some users of CICS Transaction Server and Java.

The continuous JVM was introduced in CICS Transaction Server, Version 2.3. It offers the ability to omit resetting the JVM between CICS tasks, helping to ensure that there are no performance problems due to cross-heap references.

It also offers the ability to cache states between transactions to help improve performance. As a Java application-execution environment, it is more consistent with Java in other environments and on other platforms (for example, the class loading, threading, just-in-time [JIT] and garbage-collection components are the standard ones). CICS Transaction Server is still designed to ensure complete isolation between concurrently running tasks that run in different JVMs in both JVM modes, but isolation issues might exist between serial tasks that are running in the same JVM. In practice, most CICS Java workloads now use the continuous JVM to benefit from the considerable performance advantages. With the new release of CICS Transaction Server, Version 3.2, the focus is now firmly on continuous JVM and the operational advantages that this feature confers.



Java Virtual Machine support in CICS

In this chapter, we discuss Java Virtual Machine (JVM) support in CICS. In particular, we discuss the various modes of operation that a JVM can operate in and the implications of choosing one of those modes. We look at the Java class cache and how it can reduce the JVM size and optimize class loading times. Also, we discuss different categories of classes that are loaded into a JVM, the rationale behind having those categories, and related performance and application design issues.

2.1 Overview

Java applications running in a server environment, such as CICS Transaction Server, have characteristics that are quite different from client applications. The latter tend to be long-lived, performing a variety of tasks (for example, a GUI application, such as Rational® Application Developer). Server applications, on the other hand, typically perform a narrow set of functions in a more “predictable” way, and they are designed for throughput and tend to be short-lived.

Also, in a transaction processing system, it is vital that no transaction effects the outcome of other transactions that run in parallel or subsequently, other than by updating resources under transactional control, which did not lend itself well to the JVM concept. The first problem is that the cost of creating and initializing a JVM is very high. Therefore, if a JVM was started and destroyed for each invocation of a transaction program, this leads to an unacceptable throughput rate.

The obvious solution to this problem is to start up and initialize the JVM once, and then use it to run multiple programs sequentially. This is the approach in today’s Web application servers, such as IBM WebSphere Application Server.

There is a new problem with this approach, however. A misbehaved program might change the state of the JVM in such a way that the outcome of subsequent program invocations on that JVM can be affected (a rather extreme example is a change to the current time zone). In a transaction processing system, such situations must be avoided at all cost.

One way to get around this problem is to carefully review all application programs, making sure that they are “well-behaved.” Some global state changes might be all right, or even desired, such as reading configuration information that can be accessed by the next invocation of the same program without having to read it again. However, behavior, such as the reliance of static initialization of fields in a class, can bring unexpected application activity.

Further benefits are made when sharing classes across JVMs with the Java class cache, which has two benefits: First, it reduces memory footprint, and second, it further reduces the JVM startup cost (in terms of both CPU and I/O) because shared classes are loaded and initialized only once—the next JVM to start up can then use the shared version.

2.2 History of JVM support in CICS

In the next few sections, we review the history of JVM support in CICS, starting at CICS TS 1.3 and going through to the current version CICS TS 3.2.

2.2.1 CICS Transaction Server 1.3

CICS TS 1.3 was the first level of CICS Transaction Server to support Java, at the JDK™ 1.1 level. Initially, however, Java programs were not run under a JVM; instead, they were compiled using the High Performance Java (HPJ) product, which produces a *Java Program Object*. HPJ took the Java bytecodes and compiled them to S/390 machine code. The programs then ran natively in the control of CICS Language Environment®, just like a C application. Obviously, with this approach you lose platform independence.

Because the startup cost for an HPJ compiled application is very high, much like that of a JVM, a later PTF for CICS TS 1.3 introduced a technique called *Hot Pooling*. With Hot Pooling enabled, the HPJ environment is saved and reused between applications.

JVM support (for JDK 1.1 only) was introduced with another PTF to CICS TS 1.3. Because the JDK 1.1 JVM was not designed for reuse, a JVM was initialized, used, and then terminated for each CICS JVM program, which made it very expensive in terms of path length. Realistically this single use JVM was only good for programs that were either run very infrequently or were long-lived.

JVM support was only provided in CICS TS 1.3 to provide the ability to execute those parts of the Java API that were not supported by HPJ, to provide Java compliance.

2.2.2 CICS Transaction Server 2.1 and 2.2

These two releases of CICS support the Java JDK 1.3. (The JDK 1.1 JVM is no longer supported.)

HPJ and Hot Pooling (discussed in 2.2.1, “CICS Transaction Server 1.3” on page 18) are still supported in CICS TS 2.1 and CICS TS 2.1, but only for migration. Now, you are encouraged to migrate to JVM (which really must not be more than setting up the JVM environment and re-deploying your application as Java bytecodes instead of Java Program Objects).

Support for the resettable JVM (described in 2.3.3, “Resettable JVM” on page 23) was introduced.

One restriction for CICS programs running under the JVM, in these two releases, is that a JVM program cannot link, either directly or indirectly, to another JVM program in the same CICS region, which was true for CICS TS 1.3 too.

2.2.3 CICS Transaction Server Version 2.3

This level of CICS supports JDK 1.4.1 of the IBM Software Developer Kit for z/OS, Java 2 Technology Edition. It has support for the continuous JVM (see 2.3.2, “Continuous JVM” on page 21) and introduces the shareable class cache (see 2.5, “The shared class cache” on page 27), which allows for classes and the results of JIT compilation to be shared between several JVMs.

Also, the restriction mentioned in 2.2.2, “CICS Transaction Server 2.1 and 2.2” on page 19, was lifted. You can now have more than one JVM program in the LINK stack, meaning that one JVM program can LINK to another JVM program (directly or indirectly).

2.2.4 CICS Transaction Server 3.1

CICS TS 3.1 provides new Web Services capabilities to extend CICS applications to a service-oriented architecture (SOA), enabling them to be exposed directly as Web services.

Also, JCICS support was added for the new Channels and Containers mechanism for inter-program data transfer. See 6.6.4, “Communicating through Channels and Containers” on page 105 for an example on how to use the new API.

2.2.5 CICS Transaction Server 3.2

Java 5 support was added to CICS TS 3.2 through PTF PK59577, which gives support for both Java 1.4.2 and Java 5; however, only one level can run in a CICS region at a time. Java 1.4.2 support will be removed in the next release.

Note: Java 5 runs byte codes built on earlier releases without problem.

With Java 5 comes a simplified shared class cache mechanism, enhanced garbage collection (GC), and optimizing just in time (JIT) compiler technology.

Also introduced in CICS TS 3.2 are many features for managing JVMs inside CICS, which includes usability enhancements to the JVM profile and properties files and the ability to initialize and terminate JVMs by profile.

When using Channels and Containers in CICS TS 3.2, the container data is now placed in 64-bit storage. With the JCICS support of Channels and Containers, the JVM can store and retrieve information in 64-bit storage.

2.3 JVM operation modes

IN CICS TS 3.2 the JVM can run under one of two modes:

- ▶ *Single use mode* (2.3.1, “Single use JVM” on page 20)
- ▶ *Continuous mode* (2.3.2, “Continuous JVM” on page 21)

In earlier versions of CICS (2.3 & 3.1) a third JVM mode is supported, though a deprecation message is put out when used in CICS TS 3.1:

- ▶ *Resettable mode* (2.3.3, “Resettable JVM” on page 23)

Anyone using resettable mode must migrate to a continuous mode JVM, taking into consideration the behavioral changes.

The REUSE parameter in the associated JVM profile determines what mode a JVM runs in. Figure 2-1 illustrates the three different JVM modes.

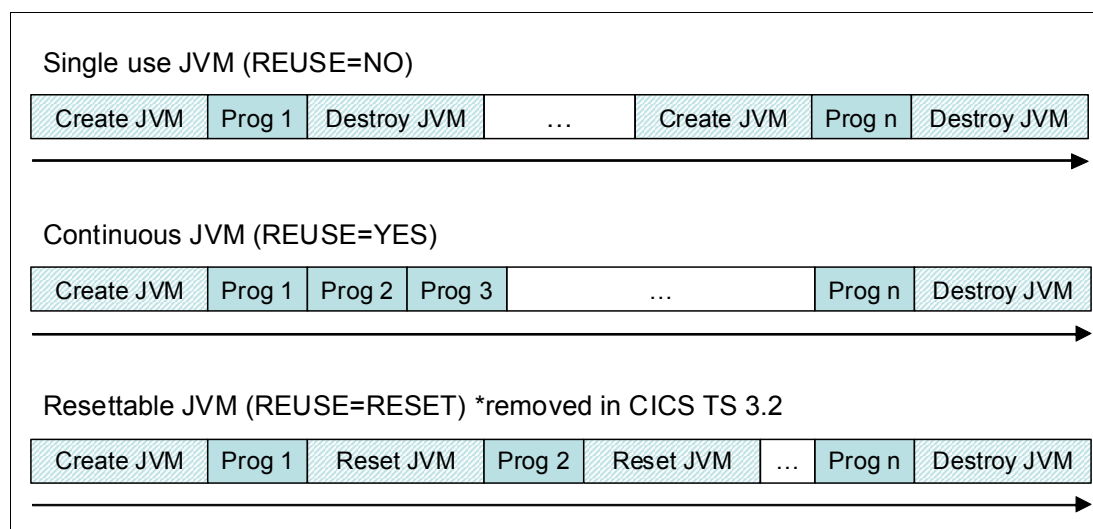


Figure 2-1 Single use, continuous mode, and resettable mode

2.3.1 Single use JVM

Single use means that for each transaction a JVM is created, the Java application is run, and then the JVM is thrown away, which effectively prevents the actions of one Java program from interfering with any other.

Enabling single use JVMs: To enable single use JVMs, add the value 'REUSE=NO' to your JVM profile.

Obviously, this mode is expensive because every application incurs the cost of starting and initializing the JVM, loading all the classes needed by it, and destroying it after the program terminated. Also, the results of JIT compilation are lost.

A typical use for this mode is testing changes to Java programs. Because no caching of classes is performed, you can be certain that when the program is restarted, the newly started JVM reloads any classes that changed on the file system. It is not recommended for use in a production environment, other than to run applications that were originally designed for single use JVMs and are not yet reviewed for suitability to run in a continuous JVM.

Single use JVMs are also required when you debug a Java program that is running in CICS. See Chapter 8, "Problem determination and debugging" on page 193 for more information about Java debugging.

2.3.2 Continuous JVM

The continuous JVM mode was introduced in CICS TS 2.3 as a way to mirror the modern behavior of Java application servers. It creates a potentially long-lived JVM, which uses a single storage heap.

Enabling continuous JVM: To enable continuous JVMs, add the value 'REUSE=YES' to your JVM profile.

In contrast to the resettable JVM mode described in 2.3.3, "Resettable JVM" on page 23, there is no mechanism for automatically resetting the state of the JVM when a Java program finishes execution.

Continuous mode offers the best performance for running Java programs in CICS; however, this comes at a price: Application designers need to be aware that actions that change the JVM's state can change the behavior of the next program (be it the same or another) that uses the same JVM. Therefore, understand the application behavior (especially when using static variables) before you deploy into a production system.

Attention: Not specifying a REUSE value in your JVM profile causes it to default to 'REUSE=YES'.

The fact that applications can pass state data can also be put to good use. Suppose, for example, an application as part of its initialization must read configuration data. This might be an expensive operation—it might involve reading DB2 tables, parsing huge XML configuration files, or doing JNDI lookups. By taking advantage of the continuous JVM, you can cache this configuration information by anchoring it in static variables, which avoids the initialization overhead on subsequent invocations of the program. Any objects that are *anchored in static*, that is, that are reachable from a static variable, are never subject to garbage collection. Example 2-1 demonstrates the technique.

Example 2-1 Caching shared configuration data

```
import java.util.*;

public class StaticLeveraging {
    private static List<String> configuration;// (1)
```

```

public static void main(String[] args) {
    if (configuration == null) populateConfigurationData();// (2)
    else System.out.println("Using cached configuration data");

    showConfigurationData();
}
private static void populateConfigurationData() { // (3)
    System.out.println("Populating with new configuration data");

    ArrayList<String> newList = new ArrayList<String>();

    // Insert some entries
    for (int i = 0; i < 3; i++) newList.add("#" + i);

    // Create a non-modifiable view of the list
    configuration = Collections.unmodifiableList(newList);// (4)
}

private static void showConfigurationData() {
    System.out.println("Values: ");
    for (String s : configuration) System.out.println "[" + s + ""];
}
}

```

Notes on Example 2-1 on page 21:

- ▶ This is the static variable where we store our configuration data.
- ▶ Check for null to see if we must initialize.
- ▶ Populate configuration data: In this example, we simply insert dummy String data. In a real application, this can be a much more expensive operation, such as reading and parsing a complex XML document.
- ▶ To make sure that the configuration data cannot inadvertently be changed by the program, we wrap them in an unmodifiable collection. In general, it is good practice to make static variables final or immutable, if possible, when programming for the continuous JVM.

When the program runs for the first time using a continuous JVM, it prints out the message in Example 2-2.

Example 2-2 Message written to stdout on first invocation

```

Populating with new configuration data
Values:
[#0]
[#1]
[#2]

```

When the program is run for a second time, the static initialization already occurred, which means that the program uses the cached data and prints out the message in Example 2-3.

Example 2-3 Message written to stdout on subsequent invocations

```

Using cached configuration data
Values:
[#0]
[#1]
[#2]

```

Be aware that there is no guarantee that subsequent executions of an application are assigned to the JVM that contains the items created by the first execution of the application. In other words, there is no guarantee that the next invocation of the program can see the data. Thus, your application must not rely on the presence of the persistent items that you create in the JVM; instead, it must check for their presence to avoid unnecessary initialization, but it must be prepared to initialize them if they are not found in the present JVM.

If you need to ensure that two applications using a continuous JVM cannot see each other's persistent state (for example, because you are not sure they will not interfere with each other), set up separate JVM profiles for the applications to use. You can specify the same options because just by using different names you make sure that they run in distinct JVMs.

2.3.3 Resettable JVM

Important: The resettable JVM mode was removed in CICS TS 3.2 and is not available in any future CICS releases. It was intended as an intermediate solution for running applications that are not known to be safe for running in continuous mode JVM. With the introduction of Java 5, the resettable mode is no longer supported at the JVM level. Therefore, you must design and code new applications that can run in continuous mode, and be prepared to migrate existing code.

2.3.4 Summary of JVM modes

Table 2-1 summarizes the different supported JVM modes.

Table 2-1 Comparison of JVM modes

	REUSE option in profile	Can program invocations pass state?	Relative performance	Compatible with the shared class cache?
Single use	NO	N/A (JVM destroyed)	Lowest (JVM initialized for each use)	No
Continuous	YES	Yes	High	Yes

2.4 Analyzing programs for use in a continuous JVM

In 2.3.2, "Continuous JVM" on page 21, we discussed the application considerations that are required when using a continuous mode JVM. The main area that needs attention is the use of static variables in a program, with special attention paid on the reliance of static initialization. To aid in the analysis of a program's suitability for running in a continuous mode JVM, you can use the CICS JVM Application Isolation Utility to produce a report on the static usage in a program, or collection of programs. Supplied as part of the CICS TS 3.2 install, or available as a SupportPac™ CH1B, the CICS JVM Application Isolation Utility scans compiled Java programs and reports any updates to static variables.

Attention: For CICS release CICS TS 2.2 and later, the utility is available as SupportPac CH1B, which you can download from the IBM CICS SupportPac site:

<http://www-1.ibm.com/support/docview.wss?rs=1083&uid=swg27007241>

2.4.1 Configuring the application isolation utility on UNIX System Services

The CICS JVM Application Isolation Utility that comes with the CICS TS 3.2 installation is located in the `/usr/lpp/cicsts/cicsts32/utls/isolation` directory (or the equivalent CICS install directory on your system) on Unix System Services. Example 2-4 shows the contents of this directory.

Example 2-4 Contents of isolation directory

<code>-rw-r--r--</code>	<code>1</code>	<code>xxxxxx</code>	<code>xxxxxx</code>	<code>2368</code>	<code>Oct 28 09:35</code>	<code>DFHIsoUtil</code>
<code>-rw-r--r--</code>	<code>1</code>	<code>xxxxxx</code>	<code>xxxxxx</code>	<code>22655</code>	<code>Oct 28 09:35</code>	<code>dfhjau.jar</code>

The file `DFHIsoUtil` is a shell script for running the utility. The file `dfhjau.jar` contains the compiled Java classes for the application. To run the `DFHIsoUtil` script, the `CICS_HOME` environment variable needs to be set to where CICS directory resides on UNIX System Services, Example 2-5 shows how to do this.

Example 2-5 Setting the CICS_HOME environment variable

```
export CICS_HOME=/usr/lpp/cicsts/cicsts32
```

When running the `DFHIsoUtil`, you make sure that it has the appropriate execution permissions. The directory listing in Example 2-4 lists the permissions of both files as `-rw-r--r--`. This means that they can both be read by everyone (user, group, and others), and modified by the user, but they cannot be executed. Because `DFHIsoUtil` is a script that we want to execute, update its permissions to look like `-rwxr-xr-x`. In Example 2-6, we show what happens when you try to run the script without execute permission, and then what happens after it is set.

Example 2-6 Setting execution permissions for DFHIsoUtil

```
./DFHIsoUtil: FSUM9209 cannot execute: reason code = ef076015: EDC5111I Permission denied.
```

```
# Now set the permissions using chmod:
chmod +x DFHIsoUtil// (1)
```

```
# List the permissions of the updated file:
ls -l DFHIsoUtil
```

```
-rwxr-xr-x 1 xxxxxx xxxxxx 2368 Oct 28 09:35 DFHIsoUtil// (2)
```

```
# Now it can be run without error:
./DFHIsoUtil
CicsIsoUtil: CICS JVM Application Isolation Utility
```

```
Copyright (C) IBM Corp. 2007
```

```
Usage: java -cp dfhjau.jar CicsIsoUtil [-options] filename [filename... filename]
Where filename is the name of a Java class or jar file
to be inspected. Multiple files may be specified, or
wildcard (glob) characters may be used.
```

```
Options may be:
```

```
-v    -verbose  enable verbose output
-?    -help     display this help text
```

Notes on Example 2-6 on page 24:

- **chmod** is a UNIX utility for modifying the permissions of files. Type `man chmod` in UNIX Systems Services to see more information about this command.
- The value `x` signifies that execute permissions were added to `DFHIsoUtil`.

Running CICS Java Application Isolation Utility program on your development platform:

Because the CICS Java Application Isolation Utility is a Java program, you can run it on your development platform (Linux, Windows, and so on):

1. Take a copy of `dfhaiu.jar` from UNIX Systems Services, and place it on your workstation using ftp binary mode transfer.
2. Making sure that Java is installed on your machine, bring up a command line window, and from the directory where the jar file now resides, type `java -classpath dfhjaiu.jar CicsIsoUtil`

2.4.2 Generating reports on static updates

With the utility configured and ready to run, you can now use it to generate reports on the updating of statics within Java programs. As we discussed earlier, you can run the program against an individual `.class` file or a collection of `.class` files that are inside a `.jar` file.

.jar file: A `.jar` file is simply a compressed file (`.zip` file) with a different extension. See this for yourself by changing the extension to `.zip` and viewing its contents.

Reporting updates to static variables

The easiest mistake to make is to rely on the static initialization of variables. Example 2-7 shows some code where the value of the `count` variable is set to 0 on static initialization of the class. In a single use JVM, this initialization occurs on every invocation of the program. However, when using a continuous JVM, static initialization happens only when the JVM is created, which means that multiple invocations of the program cause the value to continually increase. Without understanding this behavior, you might experience unexpected behavior during program execution.

Example 2-7 Updating a static variable in a program

```
public class HelloWorldStaticVariables {
    private static int count = 0; // (1)

    public static void main(String args[]) {
        count++;
    }
}
```

Notes on Example 2-7:

- `Count` gets set to 0 as part of the static initialization of the class, this only happens once, at the point the JVM is created.

Example 2-8 shows that report that running the CICS JVM Application Isolation Utility against this class produces.

Example 2-8 Report generated from the code in Example 2-7 on page 25

```
./DFHisoUtil examples/HelloWorld/HelloWorldStaticVariables.class
CicsIsoUtil: CICS JVM Application Isolation Utility
```

Copyright (C) IBM Corp. 2007

Reading class file: examples/HelloWorld/HelloWorldStaticVariables.class

```
Method: public static void main(java.lang.String[])// (1)
```

```
Static fields written in this method:
```

```
private static int count
```

```
Method: <clinit> (Class Initialization)// (2)
```

```
Static fields written in this method:
```

```
private static int count
```

```
Number of methods inspected      : 3
```

```
Total static writes for this class: 2
```

```
Number of jar files inspected    : 0
```

```
Number of class files inspected  : 1
```

The report in Example 2-8 shows that there are two places where updates are made to static variables. In case (1) the static variable count is updated inside the main() method, in case (2) the static variable is set on class initialization. Having found these instances of static updates, the code must be analyzed to determine if the behavior is problematic in a continuous JVM.

In the case of Example 2-7 on page 25, the static variable count is set to 0 only at the point when the JVM (and class) is created. On each invocation of the program within the same continuous JVM, the value of count is remembered from the last run and is incremented by 1. If this behavior is undesirable, a work around is to set count = 0 at the beginning of the main() method. However, this still causes the same report to be generated, as shown in Example 2-8.

Reporting updates to the contents of static final Objects

A safe way to use static values in a program is to make them final, which means that the static variable instance cannot be changed in the program. Adding the final modifier to static instances of primitive Java types, and immutable objects, such as instances of String, make them safe for use in a continuous JVM and so they are not reported, as shown in Example 2-8. However, this does not apply for all Java objects, for instance, even though the value of a static final object cannot be changed, this does not stop the contents of that object that is modified.

Example 2-9 on page 27 demonstrates this scenario using a *Hashtable* from the Java collections library. The Hashtable is created on static initialization of the class and is declared as final, which means that for the lifetime of this JVM the myHashtable variable will only ever point to that instance of Hashtable, that is, it is fixed and can never be changed. However, this does not stop the contents of this Hashtable being updated. Therefore this type of behavior is recognized by the CICS JVM Application Isolation Utility and reported accordingly.

Example 2-9 Updating the contents of static final objects in a program

```
public class HelloWorldStaticObjects {  
    private static final Hashtable myHashtable = new Hashtable();  
  
    public static void main(String args[]) {  
        myHashtable.put("key", "value");  
    }  
}
```

Example 2-10 shows the report that is produced when you run the CICS JVM Application Isolation Utility against this class.

Example 2-10 Using utility with a class that updates static final Object

```
./DFHIsoUtil examples/HelloWorld/HelloWorldStaticObjects.class  
CicsIsoUtil: CICS JVM Application Isolation Utility
```

Copyright (C) IBM Corp. 2007

Reading class file: examples/HelloWorld/HelloWorldStaticObjects.class

```
Method: <clinit> (Class Initialization)// (1)  
Static fields written in this method:  
    private static final java.util.Hashtable myHashtable
```

```
Number of methods inspected      : 3  
Total static writes for this class: 1
```

```
Number of jar files inspected    : 0  
Number of class files inspected : 1
```

The report in Example 2-10 shows that a static object is created during class initialization. Even though the object is declared as `final`, its contents can still be updated and cached between subsequent program invocations. As discussed in 2.3.2, “Continuous JVM” on page 21, if used correctly, this caching of information can be beneficial for retrieving reference information. However if done unintentionally, these updates to static objects can generate unpredictable program behavior.

2.5 The shared class cache

As previously discussed in 2.1, “Overview” on page 18, another potential benefit of having long-lived JVMs, other than removing JVM startup cost, is the possibility to share classes across multiple JVMs, which further cuts down the cost that is associated with loading the class files from the file system and initializing them.

The Java 5 JVM from IBM brings with it a simplified class cache mechanism over that which was supplied with Java 1.4.2, which means that you no longer have the concept of master and worker JVMs; instead, you have a single class cache that all JVMs can share. Because CICS TS 3.2 supports both Java 5 and Java 1.4.2 (though not simultaneously in the same CICS region), we discuss both class cache mechanisms in the next section.

Important: There can be, at most, one active shared class cache per CICS region.

2.5.1 Benefits of the shared class cache

The benefits of having a *shared class cache* are:

- ▶ JVMs start up more quickly.
A large proportion of JVM startup time is spent loading classes and JIT-compiling methods. With a shared class cache, classes are loaded only once.
- ▶ JVMs that use the shared class cache have lower storage requirements than standard JVMs, which means that you can have more JVMs in a single CICS region.
- ▶ The result is improved total system throughput (especially if combined with running in continuous mode).

Important: The shared class cache is no more and no less than what the name says—a cache for classes, that is, for bytecode. It is *not* a cache for data, for example, static member variables reside in one of each JVM's own heaps, not in the shared class cache.

Do not confuse this with JVMs that run in continuous mode. A continuous JVM allows sharing data across program invocations, and the shared class cache, on the other hand, does *not* allow, or render possible, sharing data across several JVMs.

2.5.2 Java 5 shared class cache

With Java 5 comes a simplified class cache mechanism. Unlike Java 1.4.2 where CICS solely managed the shared class cache, it now exists as an entity in itself, which means that it persists across CICS restarts. There are also a collection of UNIX System Services utilities for managing the Java 5 shared class cache, which we discussed in 2.5.8, “The -Xshareclasses utilities” on page 33.

Attention: Unlike the Java 1.4.2 shared class cache, the Java 5 JIT'd classes are not cached due to the new JIT optimization mechanism that is applied per JVM.

2.5.3 Java 1.4.2 shared class cache

To support the shared class cache, Java 1.4.2 introduced the concept of a JVMSet, which is comprised of one *master JVM* and a set of multiple (one or more) *worker JVMs*.

Master JVM

The sole purpose of the master JVM is to initialize and to own the shared class cache. It does not participate in any work after created; in other words, it cannot be used to run Java applications.

The JVM profile for the master JVM defines the trusted middleware classes and the shareable application classes for the shared class cache.

The name of the JVM profile for the master JVM is set up in the JVMCCPROFILE system initialization parameter.

Important: There is at most one active shared class cache, and therefore at most one master JVM per CICS region.

Worker JVMs

The worker JVMs perform the actual Java workload (Java and CORBA applications, and EJBs). For classes that are shared across JVMs, a worker JVM uses the classes that are loaded in the shared class cache instead of having to load them from the file system.

A worker JVM owns the working data for the applications that run in it (to maintain isolation between Java applications), and it also owns classes that are defined as *non-shareable*. These classes are loaded into each individual worker JVM as they are needed.

The JVM profile for a worker JVM specifies the classpath for *non-shareable* classes. All other classpath settings are ignored for worker JVMs—they are inherited from the master JVM.

Important: The worker JVMs also inherit their JVM mode from the master JVM.

2.5.4 Starting the shared class cache

The JVMCCSTART system initialization parameter configures how the shared class cache is to start. Its behavior depends on the level of Java that is used.

The various settings are:

- | | |
|------------------------|---|
| JVMCCSTART=AUTO | The shared class cache is started automatically, as soon as a JVM starts up. Also, if you stop the shared class cache manually, it is restarted as soon as a JVM needs it. |
| JVMCCSTART=YES | On a warm or emergency start, if a Java 1.4.2 shared class cache was active when the system shut down, CICS starts a new one during initialization. For any other scenario, the parameter behaves the same as AUTO. |
| JVMCCSTART=NO | You must start the shared class cache manually, using the CEMT PERFORM CLASSCACHE START command. Also, there are no automatic restarts after stopping it, which means that all attempts to create a new JVM that use the shared class cache fail until you perform the restart. |

The final step to configure a JVM to use the shared class cache is to add CLASSCACHE=YES to the JVM profile.

2.5.5 Inquiring the status of the shared class cache

To show the current status of the shared class cache, use the CEMT INQUIRE CLASSCACHE command. Figure 2-2 shows the summary display that is returned from this command. To view detailed information about a class cache, press **Tab** on the first line of the summary, and then press **Enter** to go to the view shown in Figure 2-3 on page 30.

```
I CLASSCACHE
STATUS: RESULTS - OVERTYPE TO MODIFY
  Cla Ena Sta Pro(DFHJVMCC) Dat(10/28/08) Tim(11:55:15)
    Tot(0001) Old(0000) Pha(0000) Reu Caches(24M)          )

                                SYSID=MV2C APPLID=IY0398C
RESPONSE: NORMAL                                TIME: 11.55.39 DATE: 10.28.08
PF 1 HELP          3 END          5 VAR          7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 2-2 CEMT INQUIRE CLASSCACHE panel, short form

```

I CLASSCACHE
RESULT - OVERTYPE TO MODIFY
  Classcache
  Autostartst( Enabled )
  Status(Started)
  Profile(DFHJVMCC)
  Datestarted(10/28/08)
  Timestarted(11:55:15)
  Totaljvms(0001)
  Oldcaches(0000)
  Phasingout(0000)
  Reusest(Reuse)
  Cachesize(24M          )
  Cachefree(18389160     )

                                SYSID=MV2C  APPLID=IY0398C
                                TIME:  11.56.16  DATE: 10.28.08
PF 1  HELP 2  HEX 3  END          5  VAR          7  SBH 8  SFH          10  SB 11  SF

```

Figure 2-3 CEMT INQUIRE CLASSCACHE panel, long form

The following fields are displayed:

- | | |
|-------------------------|--|
| Autostartst | Displays the status of autostart for the shared class cache (see 2.5.4, “Starting the shared class cache” on page 29). |
| Status | Displays the status of the current shared class cache. The values are: <div style="margin-left: 20px;"> <p>Started
The shared class cache is ready, and JVMs can use it. This value in the CEMT display includes both the status STARTED and the transient status RELOADING, which occurs when a CEMT PERFORM CLASSCACHE RELOAD command is issued and a new shared class cache is loaded to replace the existing shared class cache. While the shared class cache is reloading, JVMs (both those that were already allocated to tasks and those that were allocated to tasks after the command was issued) continue to use the existing shared class cache until the new shared class cache is ready.</p> <p>Stopped
The shared class cache is either not yet initialized on this CICS execution or it was stopped manually. If autostart is disabled, requests for JVMs fail. If autostart is enabled, a new shared class cache is initialized when CICS receives a request to run a Java application in a JVM whose profile requires the use of the shared class cache. This value in the CEMT display includes both the status STOPPED and the transient status STARTING, which occurs when the shared class cache is initialized, either through the autostart facility or because an explicit CEMT PERFORM CLASSCACHE START command was issued. While the shared class cache is starting, JVMs that require the use of the shared class cache wait until the startup process is complete and the shared class cache is ready. If initialization of the shared class cache is unsuccessful, any waiting requests for JVMs fail.</p> </div> |
| Cachefree(value) | Displays the amount of free space in the shared class cache, in bytes. |
| Cachesize(value) | Displays the size of the shared class cache, in bytes, kilobytes (K), megabytes (M), or gigabytes (G). If the status of the shared class |

cache is STOPPED, this is the size that is used by default when the shared class cache is started. If the status of the shared class cache is STARTING or STARTED, this is the size of the current shared class cache. If the status of the shared class cache is RELOADING, this is the size of the new shared class cache that is loaded.

Datestarted	Displays the date on which the current shared class cache was started. The format of the date depends on the value that you selected for the DATFORM system initialization parameter for your CICS region.
Oldcaches(value)	Displays the number of old shared class caches that are still present in the region because they are waiting for worker JVMs that are dependent on them to be phased out. If the status of the current shared class cache is STOPPED, and worker JVMs are still dependent on it, then that shared class cache is included in the number of old shared class caches.
Phasingout(value)	Displays the number of worker JVMs that are dependent on an old shared class cache, and are being phased out. If the status of the current shared class cache is STOPPED, then any worker JVMs that are still dependent on it are included in the number of worker JVMs being phased out.
Profile(value)	Java 5 does not use worker JVMs, so this value remains empty in all cases. For Java 1.4.2, if the status of the shared class cache is STOPPED, this displays the eight-character name of the JVM profile that will be used for a master JVM to start the shared class cache. If the status of the shared class cache is STARTED, STARTING, or RELOADING, this displays the eight-character name of the JVM profile that was used for the last valid request to start or reload the shared class cache. This name is displayed even if the shared class cache fails to start or reload. The displayed JVM profile is used the next time that you issue the command to start or reload the shared class cache, unless you specify a different JVM profile using the Profile option.
Reusest	<p>Displays the level of reusability for the JVM that initializes the shared class cache. Recall from “Master JVM” on page 28, that all of the worker JVMs in a CICS region inherit their level of JVM mode from the master JVM. The values are:</p> <p>Reuse The JVMs are running in continuous mode.</p> <p>Unknown The JVM mode is not known because the shared class cache is not started.</p>
Timestarted	Displays the time as an absolute value measured from midnight, that the current shared class cache was started. The time is in the format hh:mm:ss.
Totaljvms(value)	Displays the number of JVMs in the CICS region that are dependent on a shared class cache, which includes both the JVMs that are dependent on the current shared class cache and any JVMs that are dependent on an old shared class cache and are being phased out.

2.5.6 Changing the size of the shared class cache

When the JVM initializes the shared class cache, its size is set to the value that is specified in the JVMCCSIZE SIT parameter. To inquire the size of a running shared class cache, use the CEMT INQUIRE CLASSCACHE command, as explained in 2.5.5, “Inquiring the status of the shared class cache” on page 29.

When you find the shared class cache is too small, you can change its size by using the following commands. (Use START if the class cache is not currently started, or RELOAD if it is).

```
CEMT PERFORM CLASSCACHE [ START | RELOAD ] CACHESIZE(number)
```

The CACHESIZE parameter gives the new size of the class cache in bytes, kilobytes, megabytes, or gigabytes (use a suffix of K, M, or G, respectively).

As usual, subsequent CICS restarts use the new value for the cache size unless you performed a COLD or INITIAL start of the CICS region.

2.5.7 Updating classes in the shared class cache

There will be occasions when you want to install a new version of an application in a running CICS region. The process for performing this update differs depending on which level of Java you are using.

In Java 5, if a class is updated on the file system, the class cache becomes automatically aware of this and loads the new instance. However, a JVM does not pick this up until it is phased out to pick up the new class changes in the cache.

In Java 1.4.2, the shared class cache will not pick up the new copy automatically, so you must tell CICS about it using the CEMT PERFORM CLASSCACHE RELOAD command.

There are three levels of “gentleness” with this command:

- ▶ CEMT PERFORM CLASSCACHE RELOAD PHASEOUT

A new shared class cache is created, but until it is ready, both currently running worker JVMs and newly created worker JVMs use the old cache. When the new cache is ready, the currently running worker JVMs are allowed to finish their work, but are then terminated. Newly created worker JVMs then use the new shared class cache.

This is the default option for the RELOAD command.

- ▶ CEMT PERFORM CLASSCACHE RELOAD PURGE

All tasks using the shared class cache are terminated using the PURGE mechanism, and then the cache is deleted.

- ▶ CEMT PERFORM CLASSCACHE RELOAD FORCEPURGE

All tasks using the shared class cache are terminated using the FORCEPURGE mechanism, and then the cache is deleted.

Tip: During development (that is, when your application is updated very frequently), it might be a good idea not to use the shared class cache at all, as suggested in 2.3.1, “Single use JVM” on page 20. Sooner or later, you will forget to reload the shared class cache and will be scratching your head wondering why the bug you spent all day finding still does not seem to be fixed. So, for development or debugging purposes, you must set up a JVM profile for a single use JVM, which does not use the shared class cache at all. Any new bytecode that is deployed to CICS is picked up by a single use JVM when it starts up.

2.5.8 The -Xshareclasses utilities

Supplied with the Java 5 JVM are a collection of utilities for managing the shared class cache. They are available from the **java** command in the UNIX System Services environment.

To see the status of a shared class cache for a running CICS region, log in to UNIX System Services (through SSH, Telnet, or rlogin) using the same ID that the CICS job is running under. Example 2-11 shows the job information for a running CICS region, which is using a shared class cache. Notice that it is running under the user ID CICSUSER.

Example 2-11 Job information for the CICS region

JOBNAME	JobID	Owner
IY0398C	JOB29376	CICSUSER

After logging into UNIX System Services using the CICSUSER ID, you can then use the **-Xshareclasses** to view the shared class cache information. Example 2-12 shows how to use the **listAllCaches** utility to view all of the shared class caches that are running under the CICSUSER ID. Notice that the utility is run using the **java** command while passing the **-Xshareclasses** parameter.

Example 2-12 Querying the list of shared caches

```
java -Xshareclasses:listAllCaches
```

Shared Cache	OS shmid	in use	Last detach time
CICS_sharedcc_IY0016C_0598031		0	Tue Oct 28 11:38:18 2008
CICS_sharedcc_IY0084C_0139280		0	Tue Oct 28 12:08:19 2008
CICS_sharedcc_IY0398C_08210		1	Tue Oct 28 15:05:41 2008

Example 2-12 shows that CICSUSER has three shared class caches in existence, though right now only one of them is in use (the others might persist between CICS restarts). CICS automatically generates the name of each shared class cache using the pattern **CICS_sharedcc_<applid>_<generation number>**. The **<generation number>** is incremented each time CICS recycles the shared class cache.

After recognizing the shared class cache that is in use by your CICS region you can query it for statistical information using the **printStats** option shown in Example 2-13.

Example 2-13 Querying a the summary statistics for a shared class cache

```
java -Xshareclasses:name=CICS_sharedcc_IY0398C_0,printStats
```

Current statistics for cache "CICS_sharedcc_IY0398C_0":

base address	= 0x22500058
end address	= 0x234FFFF8
allocation pointer	= 0x226D2198
cache size	= 16777128
free bytes	= 14842164
ROMClass bytes	= 1909056
Metadata bytes	= 25908
Metadata % used	= 1%

```
# ROMClasses      = 493
# Classpaths      = 5
# URLs            = 0
# Tokens          = 0
# Stale classes   = 0
% Stale classes   = 0%
```

Cache is 11% full

Could not create the Java virtual machine.

The result from the **printStats** command in Example 2-13 on page 33 shows the amount of storage in use by the cache and counts on the items that are held in it.

The 'Could not create the Java virtual machine' message is an extraneous internal JVM message that you can ignore.

Tip: Additional utilities are available for listing the contents of a cache, destroying a cache, and expiring unused caches. They are discussed in the developerWorks® article:

<http://www.ibm.com/developerworks/java/library/j-ibmjava4/>



Part 2

Systems Programming

Part 2 of the book is intended for system programmers who are setting up and configuring CICS to run Java applications. We list the software and hardware requirements to run Java applications in CICS. We then describe how to set up and run a very simple CICS Java application. Finally, we provide more detailed information about setting up and configuring CICS to run Java applications.



Setting up CICS to run Java applications

In this chapter, we describe the necessary tasks to set up the environment for running Java applications in CICS and some of the commands to control your CICS Java setup after it is up and running. We aim the information in this chapter at CICS systems programmers, although it might be useful background information for Java application developers too.

In the first section, we give a brief overview of how to access the z/OS UNIX shell, and then we describe how you can build and run one of the Java application samples that shipped with CICS. This section is useful if you are setting up a Java application in CICS for the first time. You can skip this section if you already have some experience with setting up a Java application in CICS.

In the second section, we provide more detailed information about setting up CICS to run Java applications. We discuss how to configure z/OS, UNIX System Services, Language Environment, and CICS, and describe some of the options that you can specify in the JVM profile and JVM properties file.

In the third section, we describe some of the commands you can use to manage your CICS Java setup after it is up and running.

3.1 Running a simple Java application in CICS

In this section, we describe how you can set up and run a simple Java application in CICS. We aim this section at systems programmers who are new to Java.

We included a brief overview of accessing the z/OS UNIX shell. If you are already familiar with this, you can go straight to 3.1.2, “Setting up the CICS sample Java application” on page 40.

3.1.1 Accessing the z/OS UNIX shell

The z/OS UNIX shell is the interactive interface to z/OS UNIX. You must access the shell to configure CICS to run Java applications, invoke shell commands and utilities, and also run shell scripts.

There are several ways to access the z/OS UNIX shell. You can choose between a UNIX-like interface, a TSO interface, and an ISPF interface. You can choose the interface with which you are most familiar and get a quicker start on z/OS UNIX.

We describe three ways to access the shell in this book, as shown in Figure 3-1. See *ABCs of z/OS System Programming Volume 9*, SG24-6989, for more information about accessing the shell and UNIX System Services in general.

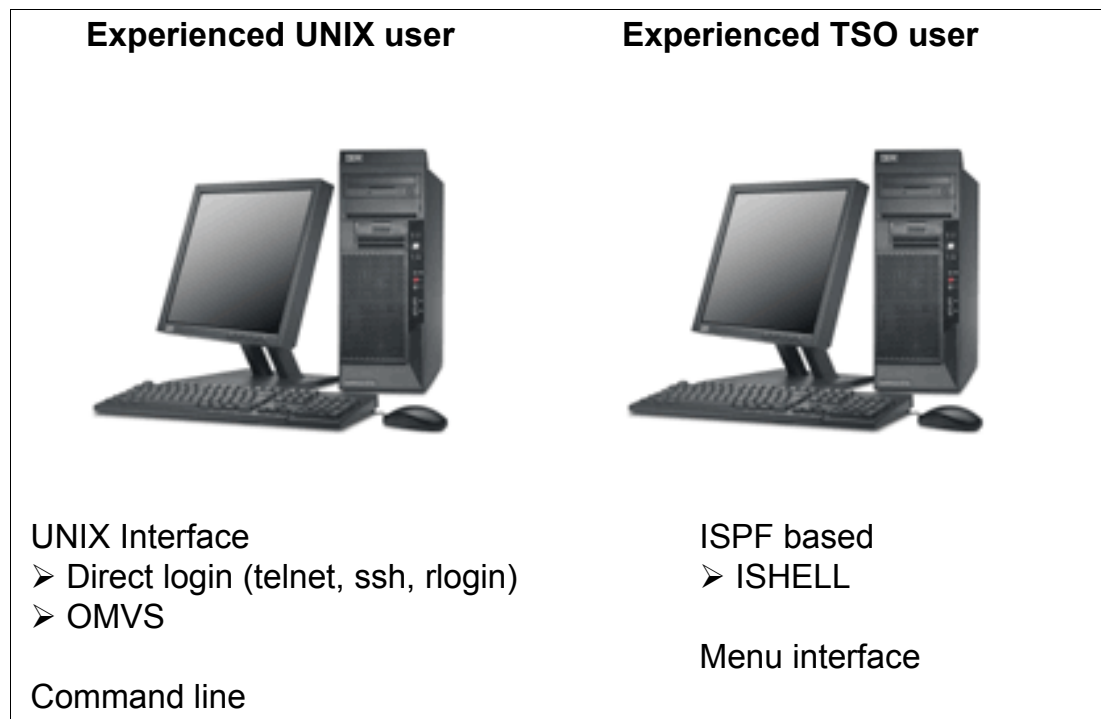


Figure 3-1 Accessing the z/OS UNIX shell

If you are an experienced TSO user and new to UNIX, we recommend that you use the ISPF shell, which is a menu-driven interface that allows you to work with the zFS and to enter some UNIX commands. You can use the ISPF editor to modify files in the zFS. If you need to type a UNIX command that is not available on the menus, then you can open a command prompt from the Tools option on the action bar (see Table 3-2 on page 45).

To start the ISPF shell, at the ISPF command prompt enter TSO ISHELL or ISHELL from ISPF option 6, as shown in Figure 3-2.

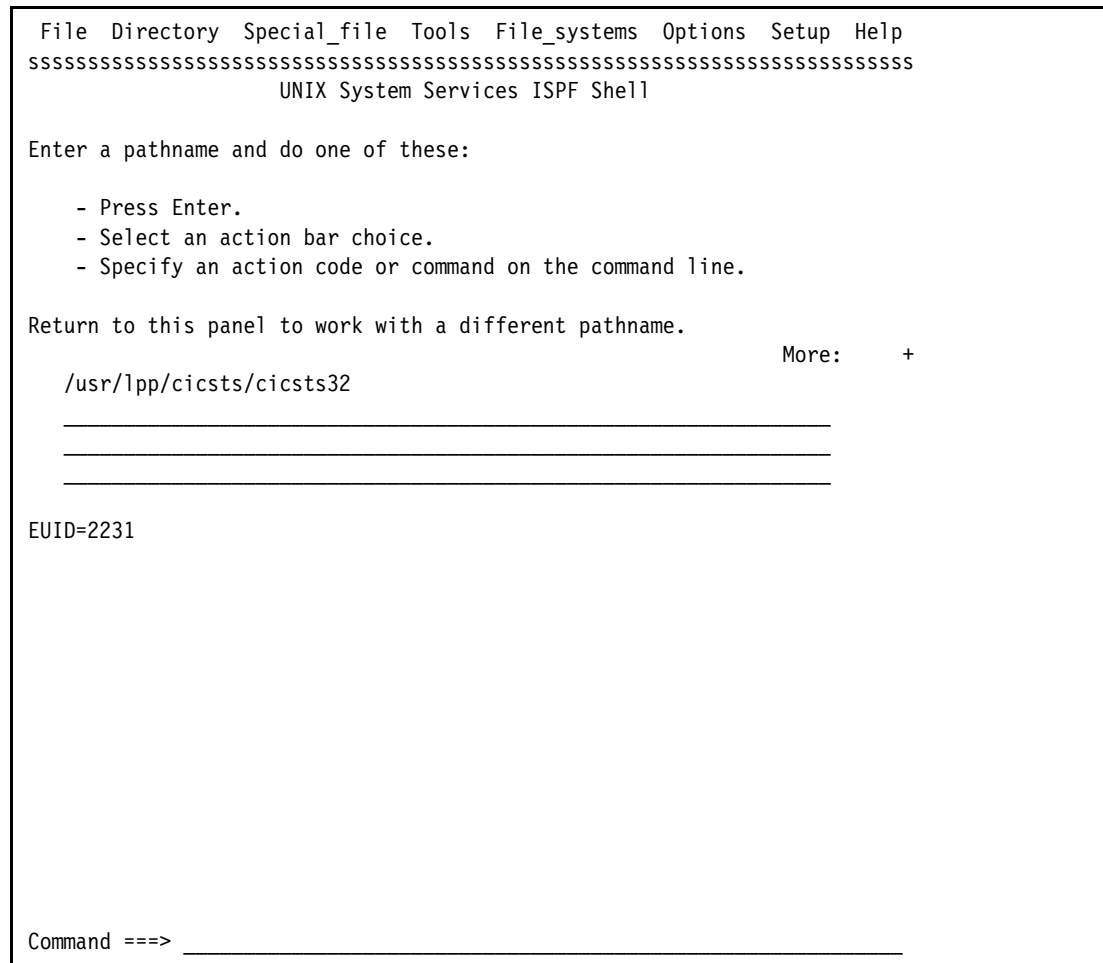


Figure 3-2 ISPF Shell

The other choice that is available from TSO is to use the OMVS command, which starts a UNIX-like interface with a command prompt. You can toggle between OMVS and the TSO/E command line. However, you cannot use the ISPF editor, nor can you use other ISPF functions, such as the split panel.

To start OMVS, at the ISPF command prompt enter TSO OMVS or enter OMVS from ISPF option 6. Figure 3-3 on page 40 shows the OMVS shell.

```

IBM
Licensed Material - Property of IBM
5694-A01 (C) Copyright IBM Corp. 1993, 2007
(C) Copyright Mortice Kern Systems, Inc., 1985, 1996.
(C) Copyright Software Development Group, University of Waterloo, 1989.

All Rights Reserved.

U.S. Government users - RESTRICTED RIGHTS - Use, Duplication, or
Disclosure restricted by GSA-ADP schedule contract with IBM Corp.

IBM is a registered trademark of the IBM Corp.

-----
Set up environment variables for Java
-----
PATH reset to /usr/lpp/java/J5.0/bin:/bin:.
JAVA_HOME reset to /usr/lpp/java/J5.0
CHRISR @ SC66:/>

====>

                                RUNNING
ESC=¢  1=Help      2=SubCmd    3=HlpRetrn  4=Top      5=Bottom   6=TSO
        7=BackScr  8=Scroll   9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve

```

Figure 3-3 OMVS shell

If you are familiar with UNIX, you might choose to log in directly to the shell. You can use the **telnet**, **rlogin**, or **ssh** command to log in. You then enter commands at the prompt. You cannot use the ISPF editor, and you cannot switch to TSO. However, you can use the TSO command to run a TSO command from your shell. If you want to modify the contents of a file, you must be familiar with a UNIX editor, such as vi or emacs.

You must know the host name or IP address of the z/OS LPAR to log in. You might also need to know the port number for the service you are using if it differs from the well known port number for that service.

We used direct login to the UNIX shell using **telnet** while writing this book. Any screen shots or figures that show the z/OS UNIX shell in this book are from a telnet session.

3.1.2 Setting up the CICS sample Java application

In this section, we provide steps to set up and run one of the CICS sample Java applications. The Java application that we selected is the HelloCICSWorld JCICS application, which is supplied with CICS TS Version 3.2. A fuller explanation of how to set up and run this and the other sample Java applications is in *Java Applications in CICS*, SC34-6825.

You:

1. Build the sample application using a makefile.
2. Create directories and files in the HFS or zFS
3. Modify the JVM profile.
4. Tailor the CICS system initialization parameters.
5. Install the required CICS resource definitions.
6. Run the Java application.

We assume that the following components are already installed, configured, and operational, and that no further configuration is necessary:

- ▶ CICS Transaction Server Version 3.2
- ▶ IBM SDK for z/OS, Java Technology Edition Version 5 Release 0
- ▶ Language Environment for z/OS
- ▶ z/OS UNIX System Services

Building the sample application using a makefile

CICS supplies makefiles that contain instructions to compile the sample Java application source code. The UNIX **make** command executes the instructions in the makefile.

You must set these three environment variables before running the **make** command:

PATH	Unix System Services search path.
CICS_HOME	The installation directory for CICS Transaction Server Version 3.2. Typically this is /usr/lpp/cicsts/cicsts32. cicsts32 is defined by the USSDIR installation parameter when you installed CICS Transaction Server Version 3.2 (cicsts32 is the default).
JAVA_HOME	The installation directory for IBM SDK for z/OS, Java Technology Edition Version 5 Release 0. In our environment, the SDK is installed in /usr/lpp/java/J5.0.

The environment variables might be set in one or more of the following places:

- ▶ In the RACF® user profile
- ▶ In the file /etc/profile
- ▶ In the file .profile in the shell user's home directory
- ▶ In the file named in the ENV environment variable in the user's .profile
- ▶ At the shell prompt
- ▶ In a shell script

These environment variables might already be set. You can use the **echo** command to check whether they were set, as shown in Example 3-1.

Example 3-1 Displaying the environment variables

```
$ [SC66] /u/cicrsr1: echo $PATH
/usr/lpp/java/J5.0/bin:bin:.
$ [SC66] /u/cicrsr1: echo $JAVA_HOME
/usr/lpp/java/J5.0
$ [SC66] /u/cicrsr1: echo $CICS_HOME

$ [SC66] /u/cicrsr1:
```

Tip: In the UNIX shell, prefix the environment variable with the dollar sign (\$) to substitute the value of the variable. The **echo** command then displays the values of the variables on the panel.

We already have PATH and JAVA_HOME set correctly because the commands to set them are in the file /etc/profile, as shown in Example 3-2.

Example 3-2 Commands to set the environment variables in /etc/profile

```
PATH=/usr/lpp/java/J5.0/bin:$PATH
export PATH
export JAVA_HOME=/usr/lpp/java/J5.0
```

Note: Example 3-2 shows two ways to export the environment variable. You can set the variable and then export it using two separate commands, or you can set and export the variable using one command.

Exporting a variable makes it available to the current process (in this case the shell) and any other processes that were created by the current process.

The null response to echo \$CICS_HOME shows that it is not set. You can either add the commands to set and export CICS_HOME to /etc/profile or your own .profile file, or on the command line, you can enter the command export CICS_HOME=/usr/lpp/cicsts/cicsts32.

Note: Adding the command to /etc/profile affects all users. Any changes made in /etc/profile takes effect the next time a user invokes the shell.

To build the Java samples, you need write permission for the HFS or zFS directory in which the samples are stored and for its subdirectories. These directories are part of the directory structure that includes the other CICS files that were installed on HFS or zFS. Alternatively, you can copy the samples directory and its subdirectories into another directory to which you have write permissions. If you do this, use your own directory in place of the \$CICS_HOME/cicsts32/samples directory in the following steps:

1. Type cd \$CICS_HOME/samples/dfjcics to change directory.
2. Type make -f HelloWorld.mak jvm to build the sample.

Example 3-3 shows the code for building the sample application.

Example 3-3 Building the sample application

```
$ [SC66] /u/cicrsr1: export CICS_HOME=/usr/lpp/cicsts/cicsts32
$ [SC66] /u/cicrsr1: cd $CICS_HOME/samples/dfjcics
$ [SC66] /cicsts32/samples/dfjcics: make -f HelloWorld.mak jvm
javac -deprecation -classpath ./usr/lpp/cicsts/cicsts32/lib/dfjcics.jar
examples/HelloWorld/HelloWorld.java
javac -deprecation -classpath ./usr/lpp/cicsts/cicsts32/lib/dfjcics.jar
examples/HelloWorld/HelloCICSWorld.java
$ [SC66] /cicsts32/samples/dfjcics:
```

You created two Java classes, HelloWorld and HelloCICSWorld that are stored in the \$CICS_HOME/samples/dfjcics/examples/HelloWorld directory.

Creating directories and files in the HFS or zFS

In this section, you copy the supplied JVM profile and JVM properties file that the HelloCICSWorld application uses to a newly created directory. JVM profiles and JVM system properties are explained in more detail in “Setting up JVM profiles” on page 49, and “JVM system properties files” on page 51. Note that, in CICS Transaction Server Version 3.2, the

JVM system properties files are optional, so you can omit the steps to create the props directory and to copy the sample properties files.

You also create a new working directory for the JVM. The CICS JVM uses this directory when creating the stdin, stdout, and stderr files for the JVM. The working directory is specified in the JVM profile.

First you create new directories under the CICS region user ID's home directory to hold the JVM profiles and properties files. You also create a new working directory. Example 3-4 shows the commands that we entered.

Example 3-4 Creating the new directories and copying files

```
$ [SC66] /u/cicsts32: export CICS_HOME=/usr/lpp/cicsts/cicsts32
$ [SC66] /u/cicsts32: mkdir JVMProfiles
$ [SC66] /u/cicsts32: cd JVMProfiles
$ [SC66] /u/cicsts32/JVMProfiles: cp $CICS_HOME/JVMProfiles/* .
$ [SC66] /u/cicsts32/JVMProfiles: cd ..
$ [SC66] /u/cicsts32: mkdir props
$ [SC66] /u/cicsts32: cd props
$ [SC66] /u/cicsts32/props: cp $CICS_HOME/props/* .
$ [SC66] /u/cicsts32/props: cd ..
$ [SC66] /u/cicsts32: mkdir workdir
```

Tip: The `cp` commands in Example 3-4 specify a period (.) as the target directory. In UNIX, this denotes the current directory.

Modifying the JVM profile

You need to check and possibly modify the values shown in Table 3-1 in the sample JVM profile DFHJVMPR.

Table 3-1 DFHJVMPR options

Option	Description	What we coded
WORK_DIR	Working directory for JVM	WORK_DIR=/u/cicsts32/workdir
CICS_HOME	CICS TS installation path	CICS_HOME=/usr/lpp/cicsts/cicsts32
JAVA_HOME	SDK installation path	JAVA_HOME=/usr/lpp/java/J5.0/
JVMPROPS	Path and name of JVM properties file	JVMPROPS=/u/cicsts32/props/dfjjvmpr.props
CLASSPATH_SUFFIX	Directory search path for Java classes	CLASSPATH_SUFFIX=/usr/lpp/cicsts/cicsts32/samples/dfjcics

If you are using ISHELL to access the shell, then you can use the ISPF editor to update the profile. If you are using OMVS or direct login to the shell, you must use a UNIX editor, such as `vi`.

Tailoring the CICS system initialization parameters

You need to add the system initialization parameter `JVMPROFILEDIR`. This is the path to the directory that holds the JVM profiles. Remember that case is significant when coding this parameter. We coded `JVMPROFILEDIR=/u/cicsts32/JVMProfiles`.

Restart CICS to pick up the change.

Installing the required CICS resource definitions

After CICS is back up, you can log on and install the CICS resource definitions.

The definitions for the CICS Java sample applications are in group DFH\$JVM. The transaction for the HelloCICSWorld Java application is JHE2 and the program is DFJ\$JHE2.

Use CEDA to install the group.

Running the Java application

Enter JHE2 at your CICS terminal. Example 3-5 shows the response.

Example 3-5 Running JHE2

JHE2Hello from a Java CICS application

The HelloCICSWorld class uses the JCICS Task class to send text to the terminal.

Note: The first time that you run JHE2, it takes a second or two to complete because of the time spent initializing the JVM. This delay does not occur the second time that you run the transaction. The JVM is reused.

You just saw one of the benefits of using a reusable JVM.

3.2 System configuration

In this section, we look at tailoring your UNIX System Services, Language Environment and the CICS Transaction Server configuration that is required to support CICS Java applications.

Java application programs are executed inside a Java Virtual Machine (JVM) within CICS.

A JVM in CICS runs as a UNIX System Services process in an Language Environment enclave created using the Language Environment pre-initialization module, CEEPIPI, which means that you must review and possibly modify your UNIX System Services and Language Environment configuration in addition to tailoring your CICS configuration to support running Java applications in CICS.

3.2.1 UNIX System Services

You specify UNIX System Services parameters in the BPXPRMxx members of SYS1.PARMLIB. The default values might not be sufficient when using the CICS JVM. *Installation Guide*, GC34-6812, suggests that you start with the following values and adjust them based on your experiences.

IPCSEMNSEMS	1000
MAXPROCSYS	500
MAXPROCUSER	512
MAXUIDS	500
MAXASSIZE	2000000000
MAXFILEPROC	512
MAXPTYS	256

MAXTHREADS 10000
MAXTHREADTASKS 5000
MAXCPUTIME 2147483647

z/OS V1R9.0 UNIX System Services Planning, GA22-7800 discusses the settings of some of these parameters. A description of all of the BPXPRMxx parameters are in *z/OS MVS Initialization and Tuning Reference*, SA22-7592.

You can change these values dynamically without re-IPLing using the SETOMVS or SETOMVS commands, for example: SETOMVS MAXPROCUSER=256

You must edit the BPXPRMxx member to pick up the changes on the next IPL.

Pay special attention to the setting of MAXPROCUSER if you have several CICS regions with the same region user ID running multiple JVMs. There is a large number of UNIX processes that run under the CICS region user ID. You might need to increase MAXPROCUSER further. Message BPXI040 is issued when the process limit is reached.

You set system-wide limits when you specify the values in BPXPRMxx. You might want to set a lower system-wide limit and then set a higher limit for individual users (such as your CICS region user ID). You can do this by specifying limits in the OMVS segment of the user's profile. Table 3-2 shows the RACF OMVS segment attribute names and the corresponding BPXPRMxx parameter names.

Table 3-2 RACF OMVS segment attributes

RACF OMVS segment attribute name	BPXPRMxx parameter name
ASSIZEMAX	MAXASSIZE
CPUTIMEMAX	MAXCPU
FILEPROC	MAXFILEPROC
PROCUSER	MAXPROCUSER
THREADS	MAXTHREADS

Use the RACF ADDUSER or ALTUSER command to specify these attributes on a per-user basis:

ALTUSER userid OMVS(PROCUSERMAX(nnnn))

3.2.2 Language Environment

Language Environment is a prerequisite for CICS JVM programs. However, unlike other languages, JVM programs do not require the CICS Language Environment interface. JVM programs run with Language Environment support using MVS services (not CICS services). JVM programs require the Language Environment support that is provided by the SCEERUN and SCEERUN2 libraries only.

You can either define SCEERUN and SCEERUN2 in the CICS STEPLIB or include them in the MVS linklist. You also must add SCEERUN to DFHRPL.

You can get the Language Environment runtime options for the enclave in which the JVM runs from:

- ▶ CEEDEOPT
- ▶ DFHJVMRO
- ▶ CICS JVM Launcher program

The options in Example 3-6 cannot be overridden and are set by the CICS JVM launcher program.

Example 3-6 Language Environment runtime options that cannot be over-ridden

```
POS(ON)
XPLINK(ON)
ABTERMENC(ABEND)
```

You can override all other runtime options for the JVM by assembling and linking a module called DFHJVMRO. The source code for a sample DFHJVMRO is provided in SDFHSAMP. A DFHJVMRO load module is not shipped.

Note: When coding options in DFHJVMRO, remember to add a trailing blank.

If you assemble and link a new version of DFHJVMRO while CICS is running, you must use the command CEMT SET PROG(DFHJVMRO) NEW to load the new version of the program.

If DFHJVMRO cannot be loaded by the JVM launcher program, CICS builds the default runtime options programmatically. Example 3-7 shows the default runtime options.

Example 3-7 Default Language Environment runtime options for the CICS JVM

```
ALL31(ON)
LIBSTACK(8,900,FREE)
STACK(128K,128K,ANY,KEEP,128k,128k)
HEAP(4M,1M,ANY,FREE,OK,4080)
BELOWHEAP(4096,2048,FREE)
ANYHEAP(4K,8176,ANY,FREE)
STORAGE(NONE,NONE,NONE,OK)
MSGFILE(CEEMSG)
ENVAR("_EDC_STOR_INITIAL=4K")
TRAP(ON,NOSPIE)
```

3.2.3 CICS Transaction Server

In this section, we describe the tasks to configure CICS to run Java applications.

Security settings for the CICS region user ID

To create JVMs, CICS requires access to directories and files in the UNIX System Services HFS or zSF. You must give your CICS region user permission to access these resources in the HFS or zFS, which is covered in detail in “Setting up Java Support and JVMs” in *Java Applications in CICS*, SC34-6825. We summarize the key points in this section:

- ▶ Assign a suitable UNIX user identifier (UID) and suitable UNIX group identifier (GID) for your CICS region user.
- ▶ Specify a home directory for your CICS region user.

- ▶ Ensure that the zFS that contains the home directory is mounted. We recommend that you use the automount facility of UNIX System Services. Refer to *z/OS V1R9.0 UNIX System Services Planning*, GA22-7800, for further information about the automount facility.
- ▶ Give the CICS region user read and execute access to the directories and files that every CICS region needs to create JVMs. These files and directories are in \$CICS_HOME and its sub-directories and in /usr/lpp/java/J5.0/bin and /usr/lpp/java/J5.0/bin/classic.
- ▶ Give the CICS region user read, write, and execute access to the working directories that you specified for stdin, stdout, and stderr for the JVMs in each CICS region. The WORK_DIR option in the JVM profile specifies the working directory. If you redirect the output using the USEROUTPUTCLASS option in the JVM profile, the CICS region user needs access to the directories to which the output is redirected.
- ▶ Give the CICS region user read and execute access to directories and files that you:
 - Specified on the CLASSPATH_SUFFIX option in the JVM profile.
 - Added to the LIBPATH in the JVM profile.
 - Are holding any JVM profiles or JVM properties files you created.

CICS startup JCL

Add the SDFJAUTH library to STEPLIB of your CICS startup JCL. This library must be APF authorized. The load modules in SDFJAUTH contain the CICS Java launcher code.

The storage that the CICS JVM uses does not come from the CICS Dynamic Storage Areas (DSAs). You must specify a region size for your CICS region that is sufficient both to accommodate the CICS DSAs and the storage that is required for your planned Java workload. *CICS Performance Guide*, SC34-6833, contains guidance on calculating and tuning the storage requirements for your Java workload. As a starting point, we recommend that you set a region size of at least 400 MB.

CICS system initialization parameters

Review the CICS system initialization parameters that we discussed in this section when you set up CICS to run Java applications.

There are some additional system initialization parameters that control CICS tracing for the JVM. You might need to set these parameters for debugging purposes. We discuss these parameters in Chapter 8, “Problem determination and debugging” on page 193.

There is more information about the system initialization parameters in *CICS System Definition Guide*, SC34-6813.

JVMPROFILEDIR

JVMPROFILEDIR specifies the name (up to 240 characters long) of an HFS or zFS directory that contains the JVM profiles for CICS. CICS searches this directory for the profiles that it needs to configure JVMs. The default value of JVMPROFILEDIR is /usr/lpp/cicsts/cicsts32/JVMProfiles.

If you chose a different name during CICS installation for the cicsts32 directory beneath which the sample JVM profiles are stored (that is, if you chose a non-default value for the CICS_HOME variable used by the DFHIJVMJ job), or if you want CICS to load the JVM profiles from a directory other than the samples directory, do one of the following:

- ▶ Change the value of the JVMPROFILEDIR system initialization parameter.
- ▶ Link to your JVM profiles from the directory that JVMPROFILEDIR specified using UNIX soft links. Using this method, you can store your JVM profiles in any place in the HFS or zFS file system. Use the `ln -s` command to create a soft link.

JVM profiles DFHJVMPR and DFHJVMCD:

The JVM profiles DFHJVMPR and DFHJVMCD and their associated JVM properties files, must always be available to CICS:

- ▶ DFHJVMPR is used if a Java program is defined as using a JVM, but no JVM profile is specified, and it is used for sample programs.
- ▶ DFHJVMCD is used by CICS-defined programs, which includes the default request processor program, the program that CICS uses to publish and retract deployed JAR files, and the program that CICS uses to manage the Shared Class Cache.

Both of these JVM profiles must therefore be present in the directory that is specified. If JVMPROPS is not specified, the properties files are not required.

JVMCCPROFILE

JVMCCPROFILE specifies the JVM profile to be used for the Java 1.4.2 master JVM that initializes the Shared Class Cache. It is not used when running Java 5.

JVMCCSIZE

JVMCCSIZE specifies the size of the Shared Class Cache on an initial or cold start of CICS. The size of the Shared Class Cache can be between 1 MB and 2047 MB. You can specify the number:

- ▶ In bytes
- ▶ As a whole number of kilobytes followed by the letter K
- ▶ As a whole number of megabytes followed by the letter M

The default value is 24 MB (specified as 24M). You can use the CEMT PERFORM CLASSCACHE START or RELOAD command (or the equivalent EXEC CICS command) to change the size of the Shared Class Cache while CICS is running.

Attention: On subsequent restarts, the value from the last CICS execution is used unless you provide JVMCCSIZE as a SIT override.

JVMCCSTART

JVMCCSTART determines whether the Shared Class Cache is started during CICS initialization and sets the status of autostart for the Shared Class Cache. The Java 5 Shared Class Cache persists across CICS restarts, so this parameter only applies if the cache was destroyed.

When autostart is enabled for the Shared Class Cache, if the Shared Class Cache was stopped or is not yet started, it is started when CICS receives a request to run a Java application in a JVM whose profile requires the use of the Shared Class Cache.

When autostart is disabled, the Shared Class Cache can only be started by a CEMT PERFORM CLASSCACHE START command (or the equivalent EXEC CICS command).

Using CEMT, you can change the status of autostart while CICS is running.

Attention: If you do this, subsequent CICS restarts use the changed setting unless the system is INITIAL or COLD started or the JVMCCSTART system-initialization parameter is specified as an override at startup. In these cases, the setting from the system-initialization parameter is used.

MAXJVMTCBS

CICS JVMs run on their own TCBs. MAXJVMTCBS limits the total number of TCBs in the pool of open TCBs that CICS uses for JVMs. Each JVM runs on a J8 or J9 TCB, so MAXJVMTCBS limits the number of JVMs that can be active in the CICS region.

The default is five. The minimum permitted value is 1, meaning that CICS can always create at least one open TCB for a JVM to use, of either J8 or J9 mode.

In Chapter 9, “Performance for Java in CICS Transaction Server Version 3” on page 221, we provide information about setting MAXJVMTCBS. Refer to *CICS Performance Guide*, SC34-6833, for definitive guidance on setting and tuning your setting of MAXJVMTCBS.

You can change the setting for MAXJVMTCBS without restarting CICS by using the CEMT SET DISPATCHER MAXJVMTCBS command.

Note: JM TCBs, used to initialize and terminate the Shared Class Cache, do not count towards the MAXJVMTCBS limit.

Setting up JVM profiles

The JVM is started by the CICS Java launcher, which uses a set of options known as a JVM profile. A JVM profile determines the characteristics of a JVM. You specify the JVM profile that a Java application uses in the CEDA PROGRAM definition for that Java program. You can set up several JVM profiles that use different options to cater to the needs of different applications. JVM profiles are text files that are stored on HFS or zFS.

The IBM JVM that CICS uses has a set of standard options that are supported in the z/OS Runtime Environment. An additional set of nonstandard options is specific to the z/OS virtual machine implementation. The IBM SDK for z/OS platforms, Java Technology Edition Version 5.0 SDK Guide contains detailed descriptions of all the options that you can specify.

Tip: If you need to set environment variables to control the behavior of the CICS JVM, specify them in the JVM profile.

Typically, you set options in the profile that specifies:

- ▶ Whether the JVM runs in continuous, or single-use mode
- ▶ Whether the JVM uses the Shared Class Cache or not
- ▶ Directory path names, such as CICS_HOME and JAVA_HOME
- ▶ The location of the JVM systems properties file, if there is one
- ▶ The Class paths for application classes
- ▶ Stack and heap size settings for the JVM
- ▶ Where output from the JVM (System.out and System.in) is written

Tip: You can use the symbol &APPLID; in any value to indicate that the APPLID of the CICS region must be substituted at runtime. This symbol allows the use of the same profile for all regions, even if a different WORK_DIR (for example) is required. APPLIDs are always upper case. You can also use the symbol &JVM_NUM; in any value to indicate that the unique JVM number, which is its UNIX System Services Process Id (pid) must be substituted at runtime. This allows for JVM-unique file naming.

CICS System Definition Guide, SC34-6813, lists the options that are of particular relevance to CICS. Table 3-3 on page 50 shows the options that you are most likely to set. Each entry in the table shows the default value if not specified in the profile. It also shows, for each type of JVM, whether the option is required (must be specified), OK (might be specified), or ignored

(CICS ignores the option if specified). If a particular setting for the option is required for a certain type of JVM (Classcache or Standalone) or if the option is only applicable to a certain type of JVM, this information is given instead.

Refer to *CICS System Definition Guide*, SC34-6813 and IBM SDK for z/OS platforms, Java Technology Edition Version 5.0 SDK Guide, for descriptions of the options.

Table 3-3 JVM profile options

Option	Default	Classcache	Standalone
CLASSCACHE	NO	YES	NO
REUSE	YES	YES or NO	YES or NO
CICS_HOME	None	Required	Required
JAVA_HOME	None	Required	Required
WORK_DIR	/tmp	OK	OK
CLASSPATH_SUFFIX	None	OK	OK
LIBPATH_SUFFIX	None	OK	OK
JVMPROPS	None	OK	OK
DISPLAY_JAVA_VERSION	NO	YES or NO	YES or NO
GC_HEAP_THRESHOLD	85	50-100	50-100
IDLE_TIMEOUT	30	0-10080	0-10080
PRINT_JVM_OPTIONS	NO	YES or NO	YES or NO
STDERR	dfhjvmerr	OK	Ok
STDIN	dfhjvmin	OK	OK
STDOUT	dfhjvmout	OK	OK

Options to be passed to the JVM at startup are prefixed with a hyphen '-', for example, -Xmx32M sets the maximum JVM heap size to 32 megabytes. CICS does not check options that are prefixed with a hyphen; instead, CICS simply passes them to the JVM, which means that you can specify system properties for the JVM in the JVM Profile by using '-Dpropertyname=value'

PRINT_JVM_OPTIONS=YES causes every option that is passed to the JVM to be printed.

If you have a pre-CICS Transaction Server Version 3.2 JVM Profile, you can still use it, unless it specifies REUSE=RESET as the resettable mode of the JVM, which is no longer supported. Any other obsolete options are identified with a message that suggests the best replacement option.

CICS supplies sample JVM profiles in /usr/lpp/cicsts/\$CICS_HOME/JVMProfiles. You can use these profiles for your own applications, only adding your classes to the CLASSPATH_SUFFIX options. The CICS installation job, DFHIJVMJ, substitutes your values for the symbols &CICS_HOME and &JAVA_HOME in the sample JVM profiles.

The supplied sample JVM profiles are:

DFHJVMPR	Reusable JVM. Shared Class Cache is not used. This is the default JVM profile for a CICS PROGRAM definition.
DFHJVMPD	Reusable JVM using the Shared Class Cache.
DFHJVMPD	Single-use JVM. We do not recommend using this profile for Java applications running in production systems.
DFHJVMCC	Java 1.4.2 Master JVM that initializes the Shared Class Cache. It is not used when running Java 5.
DFHJVMCD	Profile for CICS-supplied system programs written in Java. Do not use this profile for your own programs. Only make changes so that it is set up correctly for your CICS region to allow the CICS-supplied programs to run, for example, you might need to change JAVA_HOME.

Java Applications in CICS, SC34-6825, contains detailed information about the CICS-supplied JVM profiles.

JVM system properties files

Each JVM profile can optionally reference a JVM properties file, which is another text file containing the system properties for the JVM. The JVMPROPS option in the profile specifies the path of the systems properties file. System properties are key name and value pairs.

IBM SDK for z/OS platforms, Java Technology Edition Version 5.0 SDK Guide, contains a list of system properties that you might need to specify. IBM Developer Kit and Runtime Environment, Version 5.0 Diagnostics Guide, SC34-6650 describes system properties that you might need to set to gather diagnostic materials when working with IBM support personnel.

Note: Any system property that you specify using the **-D** option when launching a JVM from the command line can go in the JVM systems property file, with or without the **-D** prefix or with the **-D** prefix in the JVM profile.

CICS supplies sample JVM system properties files in `/usr/lpp/cicsts/&CICS_HOME/props` to get you started. There is a corresponding system properties file for each supplied JVM profile.

Table 3-4 Supplied JVM system properties file

Supplied JVM profile name	Corresponding JVM systems property file
DFHJVMPD	dfjjvmpr.props
DFHJVMPD	dfjjvmpr.props
DFHJVMPD	dfjjvmpr.props
DFHJVMCD	dfjjvmcd.props

Note: Because JVM profiles and JVM properties files are HFS or zFS files, case is important whenever you use their names in CICS. You must enter the name using the same combination of upper and lower case characters that is present in the HFS or zFS file name.

In the CEDA panels, any field where you enter an HFS or zFS file name accepts mixed case characters. This is not true if you enter the name in a CEDA command line or in another CICS transaction, such as CEMT or CECI. Therefore, you might to temporarily suppress upper case translation for your terminal when working with HFS or zFS file names. Enter CEOT NOUCTR at a CICS terminal to do this. CEOT UCTR restores upper case translation for your terminal.

Pay attention to case when specifying HFS or zFS file names in the source for your system initialization table (SIT) or SIT overrides.

CICS PROGRAM definition attributes

You use a PROGRAM resource definition to specify the control information that CICS requires to load and run an application program. Some attributes of the PROGRAM resource definition are applicable only to Java application programs, and other attributes require fixed values for Java application programs. There is more information about PROGRAM attributes in *CICS Resource Definition Guide*, SC34-6815.

JVM

You must specify JVM(YES) for a Java application.

JVMCLASS

JVMCLASS specifies the fully qualified name of the main class in a Java program to be run in a CICS JVM. The fully qualified name is the class name qualified by the package name. The name is case-sensitive and must be entered with the correct combination of upper and lower case letters.

Tip: The package name describes a hierarchical directory structure. Specify the path to the directory structure (which might be within a Java archive (JAR) file) in CLASSPATH_SUFFIX. The JVMCLASS attribute then specifies how to locate the class within the directory structure, for example, the CICS-supplied HelloWorld class is in package examples.HelloWorld. The supplied sample creates the directory structure examples/HelloWorld below /usr/lpp/cicsts/cicsts32/samples/dfjcics. The HelloWorld class file is created in /usr/lpp/cicsts/cicsts32/samples/dfjcics/examples/HelloWorld.

So you specify /usr/lpp/cicsts/cicsts32/samples/dfjcics on your classpath and examples.HelloWorld.HelloCICSWorld as your JVMCLASS attribute in the program definition.

JVMPROFILE

JVMPROFILE specifies the name of a JVM profile to use for this program. The JVM profile must be in the directory that is specified by the system initialization parameter JVMPROFILEDIR.

LANGUAGE

This attribute is ignored for JVM programs. Specifying JVM(YES) tells CICS that this is a Java program.

CONCURRENCY

JVM programs must be defined as threadsafe.

3.3 Managing your CICS Java environment

The CICS CEMT transaction provides several commands to manage your CICS Java environment. In this section, we list and briefly describe the commands that you are most likely to use. For more information about each of the commands, refer to *CICS Supplied Transactions*, SC34-6817.

All of these commands are available in the system programming interface (SPI). You can write your own programs to use EXEC CICS commands to manage. Refer to *CICS System Programming Reference*, SC34-6820, for more information.

If you use CICSplex® System Manager (CPSM) to manage your CICSplex, some of these commands are available using the CPSM Web User Interface (WUI) or CPSM Application Programming Interface (API). There is more information about using the WUI in *CICSplex SM Web User Interface Guide*, SC34-6841.

3.3.1 CEMT INQUIRE CLASSCACHE

INQUIRE CLASSCACHE, shown in Example 3-8, returns information about the Shared Class Cache in the CICS region and reports the presence of any old Shared Class Caches that are awaiting deletion.

Example 3-8 CEMT INQUIRE CLASSCACHE

```
I CL
STATUS: RESULT - OVERTYPE TO MODIFY
  Classcache
  Autostartst( Enabled )
  Status(Started)
  Profile()
  Datestarted(04/06/05)
  Timestarted(10:08:38)
  Totaljvms(0000)
  Oldcaches(0000)
  Phasingout(0000)
  Reusest(Reuse)
  Cachesize(24M          )
  Cachefree(24M)
```

Attention: The Java 5 JVM does not provide CICS with information about how full the cache is, so the Cachefree value always shows the same as Cachesize.

You can modify the autostart status of the Shared Class Cache. Autostartst takes one of two values:

Enabled	If the Shared Class Cache was stopped or is not yet started on this CICS execution, the Shared Class Cache is started as soon as CICS receives a request to run a Java application in a JVM whose profile requires the use of the Shared Class Cache.
Disabled	If the Shared Class Cache was stopped or is not yet started on this CICS execution, an explicit CEMT PERFORM CLASSCACHE START command is required to start it. If the status of the Shared Class Cache is Stopped and autostart is disabled and CICS receives a request to run a Java application in a JVM whose profile requires the use of the Shared Class Cache, the request fails.

3.3.2 CEMT INQUIRE DISPATCHER

INQUIRE DISPATCHER returns information about the current state of the dispatcher. You can see the current number of active JVM TCBs (Actjvmtcbs). The value that is displayed includes both J8 and J9 TCBs, but not the JM TCBs that are used for Shared Class Cache managment. You can also view and update the maximum number of JVM TCBs from this panel (Maxjvmtcbs). Example 3-9 shows the syntax for CEMT INQUIRE DISPATCHER.

Example 3-9 CEMT INQUIRE DISPATCHER

```
I DIS
STATUS: RESULTS - OVERTYPE TO MODIFY
Actjvmtcbs(000)
Actopentcbs(0000)
Actssltcbs(0000)
Actxptcbs(000)
Aging( 00500 )
Maxhptcbs( 005 )
Maxjvmtcbs( 005 )
Maxopentcbs( 0130 )
Maxssltcbs( 0008 )
Maxxptcbs( 005 )
Mrobatch( 001 )
Runaway( 0015000 )
Scandelay( 0100 )
Subtasks(000)
Time( 0001000 )
```

3.3.3 CEMT INQUIRE JVM

INQUIRE JVM displays all of the JVMs in the CICS region. You cannot change the displayed values. You can also enter CEMT INQUIRE JVM with one of the values listed below to display all of the JVMs in the CICS region with a particular status, for example, you can enter CEMT INQUIRE JVM UEXECKEY to display all of the JVMs that execute in user key or CEMT INQUIRE JVM PROFILE(DFHJVMPC) to display all of the JVMs that were created with the JVM profile DFHJVMPC.

Example 3-10 CEMT INQUIRE JVM

```
I JVM
STATUS: RESULTS - OVERTYPE TO MODIFY
  Jvm(0017171306) Age(0000001810) All0(0000000000) Cla Reu
    Uex Pro(DFHJVMP) Tas(00000000)
  Jvm(0050725839) Age(0000001804) All0(0000000000)      Reu
    Uex Pro(DFHJVMP) Tas(00000000)
  Jvm(0020525217) Age(0000001711) All0(0000000000)      Reu
    Uex Pro(DFHJVMP) Tas(00000000)
```

Note: The number that is associated with the JVM is the UNIX System Services process id that is executing the JVM and can be used for diagnostic information in UNIX System Services.

3.3.4 CEMT INQUIRE JVMPOOL

INQUIRE JVMPOOL displays information about the JVM pool (if any JVMs exist) in the CICS region. There is no identifier on this command. A CICS region supports only one pool of JVMs.

You can see the number of JVMs that were initialized and are available for use or allocated to tasks. This total includes JVMs that are in the process of being terminated and removed from the region and included in the PHASINGOUT count.

You can enable or disable the JVM pool. A disabled JVM pool cannot accept new requests to service JVM programs. Programs can still execute if they were started before the JVM pool became disabled. Example 3-11 shows the CEMT INQUIRE JVMPOOL.

Example 3-11 CEMT INQUIRE JVMPOOL

```
I JVMPO
STATUS: RESULT - OVERTYPE TO MODIFY
Status( Enabled )
  Total(0001)
  Phasingout(0000)
  Terminate(          )
```

3.3.5 CEMT INQUIRE PROGRAM

INQUIRE PROGRAM returns information about an installed program definition, as shown in Example 3-12.

Example 3-12 CEMT INQUIRE PROGRAM

```
I PROG(TRADERPJ)
STATUS: RESULTS - OVERTYPE TO MODIFY
  Prog(TRADERPJ)          Jav Pro Ena      Ced
    Res(000) Use(0000000001) Bel Uex Ful Thr Cic      Jvm
```

The expanded display shows the class name and JVM profile. You can change the class name and JVM profile from this display. You might need to disable upper case translation before specifying a new class name. Example 3-13 on page 56 shows the CEMT INQUIRE PROGRAM.

Example 3-13 CEMT INQUIRE PROGRAM (expanded display)

```
I PROG(TRADERPJ)
RESULT - OVERTYPE TO MODIFY
  Program(TRADERPJ)
  Length()
  Language(Java)
  Progtype(Program)
  Status( Enabled )
  Sharestatus(          )
  Copystatus( Notrequired )
  Cedfstatus( Cdf )
  Dynamstatus(Notdynamic)
  Rescount(000)
  Usecount(0000000001)
  Dataloc(Below)
  Execkey(Uexeckey)
  Executionset( Fullapi )
  Concurrency(Threadsafe)
  Apist(Cicsapi)
  Remotesystem()
  Runtime( Jvm )
  Library()
  Librarydsn()
  Jvmclass( com.ibm.itso.sg245275.trader.SimpleTraderPL )
  Jvmclass( )
  Jvmclass( )
  Jvmclass( )
  Jvmclass( )
  Jvmprofile( DFHJVMPR )
```

3.3.6 CEMT PERFORM CLASSCACHE

You can use the PERFORM CLASSCACHE command to initialize (start or reload) or terminate (phase out, purge, or forcepurge) the Shared Class Cache. Although you are performing one of these operations, you can use other options on the command to set attributes of the Shared Class Cache:

- ▶ When you initialize the Shared Class Cache (PERFORM CLASSCACHE START or RELOAD), you can change its size (Cachesize option).
- ▶ When you are terminating the Shared Class Cache (PERFORM CLASSCACHE PHASEOUT, PURGE, or FORCEPURGE), you can set the status of autostart (Autostartst option). If you do not want the Shared Class Cache to start up again until you enter an explicit command, you can use this option to ensure that autostart is disabled.

Subsequent CICS restarts use the most recent settings that you made using the PERFORM CLASSCACHE command (or the SET CLASSCACHE command), unless the system is INITIAL or COLD started or the system initialization parameters are specified as overrides at startup. In these cases, the settings from the system initialization parameters are used. Example 3-14 on page 57 shows the CEMT PERFORM CLASSCACHE.

Example 3-14 CEMT PERFORM CLASSCACHE

PER CL

STATUS: COMMAND SYNTAX CHECK

```
CEMT Perform Classcache
( Reload | Start )
( Forcepurge | PHaseout | PUrge )
< ( Disabled | Enabled > )
< PProfile() >
< Cachesize() >
```

Use one of the following options to terminate the Shared Class Cache.

Forcepurge	All tasks that use JVMs that are dependent on the Shared Class Cache are terminated by the SET TASK FORCEPURGE mechanism, and the JVMs are terminated. No more JVMs can use the Shared Class Cache, and it is deleted when all the JVMs that were dependent on it are terminated.
Phaseout	All JVMs that use the Shared Class Cache are marked for deletion. The JVMs are actually deleted when they finish running their current Java programs. No more JVMs can use the Shared Class Cache, and it is deleted when all of the JVMs that were dependent on it were terminated.
Purge	All tasks that use JVMs that are dependent on the Shared Class Cache are terminated by the SET TASK PURGE mechanism, and the JVMs are terminated. No more JVMs can use the Shared Class Cache, and it is deleted when all of the JVMs that were dependent on it were terminated.

Note: The Phaseout, Purge, and Forcepurge operations also act on any old Shared Class Caches that are still present in the region because they are waiting for JVMs that are dependent on them to be phased out, for example, if you issue a CEMT PERFORM CLASSCACHE Forcepurge command, all tasks that use JVMs that are dependent on a Shared Class Cache are forcepurged (both those dependent on the current Shared Class Cache and those dependent on old Shared Class Caches).

Note: The Profile option refers to the Java 1.4.2 master JVM Profile. Any value that is entered there is ignored when running Java 5.

3.3.7 CEMT PERFORM JVMPOOL

You can use the PERFORM JVMPOOL command to start JVMs without running an application program. You specify the number of JVMs that are required, the execution key, and the JVM profile to be used by them.

You can also use the command to terminate all or some of the JVMs in the pool to implement profile changes or add new application classes. Example 3-15 shows the CEMT PERFORM JVMPOOL.

Example 3-15 CEMT PERFORM JVMPOOL

```
P JVMPOOL START JVMC(2) UEX JVMPROF(DFHJVMPR)
STATUS: RESULTS
```

```
Initialize( Start )
Jvmcount( 002 )
Execkey( Uexeckey )
Terminate(          )
Jvmprofile( DFHJVMPR )
```

3.3.8 CEMT SET DISPATCHER

You can use the SET DISPATCHER command to specify the maximum number of J8 and J9 mode open TCBs that can exist concurrently in the CICS region. The value specified is in the range of 1 to 999. If you reduce MAXJVMTCBS from its previously defined value, and the new value is less than the number of open TCBs that are currently allocated. CICS detaches TCBs to achieve the new limit only when they are freed by user tasks. Transactions are not abended to allow TCBs to be detached to achieve the new limit. If there are tasks that are queued waiting for a J8 mode, TCB and you increase MAXJVMTCBS from its previously defined value, and CICS attaches a new TCB to resume each queued task, up to the new limit. Example 3-16 shows the CEMT SET DISPATCHER.

Example 3-16 CEMT SET DISPATCHER

```
SET DIS MAXJVMTCBS(6)
STATUS: RESULTS - OVERTYPE TO MODIFY          NORMAL
Actjvmtcbs(000)
  Actopentcbs(0000)
  Actssltcbs(0000)
  Actxptcbs(000)
  Aging( 00500 )
Maxjvmtcbs( 006 )
  Maxopentcbs( 0130 )
  Maxssltcbs( 0008 )
  Maxxptcbs( 005 )
  Mrobatch( 001 )
  Runaway( 0015000 )
  Scandelay( 0100 )
  Subtasks(000)
  Time( 0001000 )
```

3.3.9 CEMT SET JVMPOOL

Using the SET JVMPOOL command, shown in Example 3-17, you can enable or disable the JVM pool or to terminate the pool altogether.

Example 3-17 CEMT SET JVMPOOL

```
SET JVMPO PHASE
STATUS: RESULTS - OVERTYPE TO MODIFY          NORMAL
  Status( Enabled )
  Total(0000)
  Phasingout(0000)
  Terminate( Phaseout )
```

Use one of the following options to terminate a pool:

Forcepurge	All tasks that use JVMs in the pool are terminated by the SET TASK FORCEPURGE mechanism. The Shared Class Cache is deleted when all of the JVMs that were dependent on it were terminated.
Phaseout	All JVMs in the pool are marked for deletion. The JVMs are actually deleted when they finish running their current Java program. The Shared Class Cache is deleted when all of the JVMs that were dependent on it were terminated.
Purge	All tasks that use JVMs in the pool are terminated by the SET TASK PURGE mechanism, and the JVMs are terminated. The Shared Class Cache is deleted when all of the JVMs that were dependent on it were terminated.

Tip: If you change the JVM profile or JVM system properties file for a JVM that is currently in the pool, and you want the changes to take effect, you need not terminate the whole JVM pool; instead, you can use the PERFORM JVMPOOL command to terminate only those JVMs with that profile.



Part 3

Java programming for CICS



Getting started

In this chapter, we help you get started with Java programming in a CICS environment using the latest IBM software development platform on system z, Rational Developer for System z.

We show you how to create the Java code in Rational Developer for System z and how to deploy the code to the mainframe and run it under CICS.

4.1 Coding your application in Rational Developer for System z

Let us get started creating our first JCICS application. Traditionally, the first program to write and run in a new environment prints something like `Hello World!`, and the example presented in this chapter is no exception. To code your application in Rational Developer for System z:

1. Start Rational Developer for System z, and create a new Java project by selecting **File** → **New** → **Project**, and then select **Java Project**. Click **Next**, and enter the project name SG245275. Click **Finish**.
2. Create a Java package called `com.ibm.itso.sg245275` by selecting **File** → **New** → **Package**, and then create a new Java class called `HelloWorld`.
3. We show the source code for our Hello World in Example 4-1. It is a bit longer than usual for a first-time example, but it demonstrates a lot of useful things. Copy the program code from the PDF file (or the zip file available from the Redbooks Web site), and paste it into Rational Developer for System z. You will get tons of compilation errors, but we will fix these errors in the next step.

Example 4-1 Hello World!

```
package com.ibm.itso.sg245275;

import java.io.PrintWriter;
import com.ibm.cics.server.*;

public class HelloWorld {

    public static void main(CommAreaHolder cah) {                // (1)
        try {
            PrintWriter out = getOutputWriter();                // (2)
            Task task = Task.getTask();                          // (3)
            out.println();
            out.println("Hello " + task.getUserID() + ", welcome to CICS!"); // (4)
            out.println();
            out.println("This is program " + task.getProgramName()); // (5)
            out.println("Transaction name is " + task.getTransactionName()); // (6)
            out.println("CICS region is " + Region.getSYSID()); // (7)
        } catch (Throwable e) {
            System.err.printStackTrace();                        // (8)
        }
    }

    /**
     * Return the terminal principal facility, or <code>null</code>
     * if the transaction is not associated with a terminal.
     */
    private static TerminalPrincipalFacility getTerminal() {
        Object principalFacility = Task.getTask().getPrincipalFacility(); // (9)
        if (principalFacility instanceof TerminalPrincipalFacility)
            return (TerminalPrincipalFacility) principalFacility; // (10)
        else
            return null; // (11)
    }

    /** Is the transaction associated with a terminal? */
    private static boolean haveTerminal() {                        // (12)
        return getTerminal() != null;
    }
}
```

```

/**
 * Return a PrintWriter that is either directed at the terminal, if
 * we have one, or to <code>System.out</code> if not.
 */
private static PrintWriter getOutputWriter() {
    if (haveTerminal())
        return Task.getTask().out;                // (13)
    else
        return new PrintWriter(System.out, true);  // (14)
}
}

```

Notes on Example 4-1 on page 64:

- ▶ The first thing to notice is that, unlike “regular” standalone Java applications, the `main()` method in CICS Java programs takes an argument of type `CommAreaHolder`, instead of the familiar `String[]`. A main method with a string array argument is supported as well, but if both are present, CICS calls the one with the `CommAreaHolder` argument. We talk about `Commareas` in Calling other programs and passing data.
- ▶ Get an output writer to print to. Depending on the environment, output either goes to the terminal if we have one; otherwise, it goes to a zFS file. However in CICS Transaction Server 3.2, you can get the task `PrintWriter` to associate the related terminal or `System.out` by using `Task.getTask().out` directly to make life easier. The reason we keep the `getOutputWriter()` implementation is to give you a chance to have a better understanding the JCICS API better.
- ▶ Get a singleton instance of class `Task`, which has methods to inquire about information about our environment, such as the:
 - User ID that is signed on
 - Program name
 - Transaction code under which we are running
- ▶ Print the system ID of our CICS region.
- ▶ Any uncaught exceptions are caught here, and a stack trace is printed on `System.err`. Note that we could have decided to only catch checked exceptions; if an unchecked exception (such as `NullPointerException`) occurred, launcher code in CICS catches it, which in turn causes the task to abnormally end (abend).
- ▶ Get the *principal facility* associated with the task. CICS assigns the principal facility when it initiates the task, and the task *owns* the facility for its duration. No other task can use that terminal until the owning task ends.
- ▶ Not all tasks have a principal facility. If we do have one, it is an instance of `TerminalPrincipalFacility`.
- ▶ If we do not have a terminal, return *null*.
- ▶ Return *true* if we have a terminal.
- ▶ If we have a terminal, return a *PrintWriter* whose output goes to the terminal; otherwise, return a *PrintWriter* whose output goes to `System.out`. This stream is associated with a file on the zFS file system. The boolean parameter to the `PrintWriter` constructor call causes the `println()` methods to flush the output buffer.

As we said before, you will get a lot of compilation errors because the Jar file implementing the JCICS API (dfjcics.jar) is missing from the project. Because it does not come with the Rational Developer for System z product, we must get a copy from the mainframe:

1. Connect to the remote mainframe system in the Remote Systems view in the z/OS Project Perspective, and expand the UNIX System Services Files.
2. Navigate to your CICS installation directory. the dfjcics.jar file is in the lib sub-directory. Right-click, and select **Copy**.
3. Navigate to the project root directory in Local Files, right-click, and select **Paste** to download a copy to the local hard disk, as shown in Figure 4-1.
4. Switch back to the Java perspective, right-click the project, select **Refresh**. Now the dfjcics.jar is there.

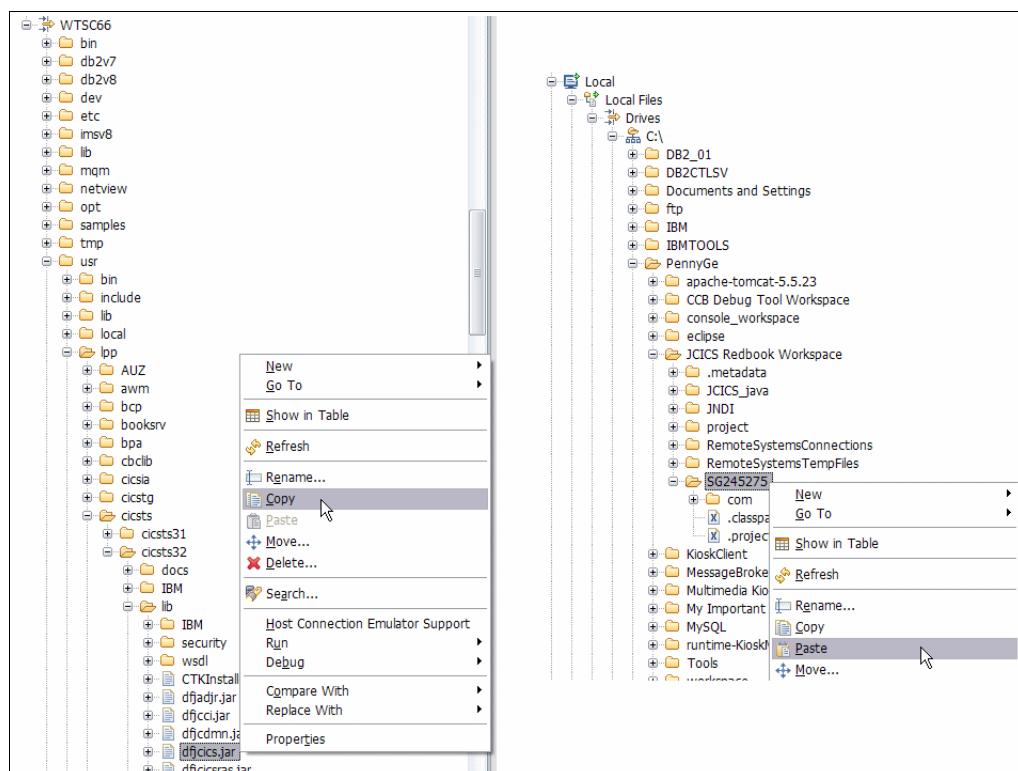


Figure 4-1 Download dfjcics.jar to the local project on the workstation

Tip: If you do not know the location in the file system of the Rational Developer for System z workspace, and therefore our project root folder, (the path name tends to be rather long), there is an easy way to find out:

1. Right-click the project, and select **Properties**.
2. In the dialog that opens, select the **Info** category. The project root folder name appears in the dialog box, and you can copy it from there, and paste it into the command prompt window.

As an alternative, you can also use the traditional way to download dfjcics.jar using FTP into your project directory if the mainframe is not set up for Rational Developer for System z yet. We list a sample FTP session in Example 4-2 on page 67 that assumes that the Rational Developer for System z workspace and the CICS library files are at their installation-default location.

Example 4-2 Sample FTP session to copy dfjcics.jar to the development workstation

```
C:\Documents and Settings\TOT195>cd C:\Documents and Settings\TOT195\IBM\rational
lsdp6.0\workspace\SG24-5275
C:\Documents and Settings\TOT195\IBM\rational\lsdp6.0\workspace\SG24-5275>ftp wtsc
66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
220-FTPMVS1 IBM FTP CS V1R6 at wtsc66.itso.ibm.com, 23:36:52 on 2005-04-05.
220 Connection will close if idle for more than 5 minutes.
User (wtsc66.itso.ibm.com:(none)): cicsrs3
331 Send password please.
Password: *****
230 CICSRS3 is logged on. Working directory is "CICSRS3.".
ftp> cd /usr/lpp/cicsts/cicsts32/lib
250 HFS directory /usr/lpp/cicsts/cicsts32/lib is the current working directory
ftp> bin
200 Representation type is Image
ftp> get dfjcics.jar
200 Port request OK.
125 Sending data set /usr/lpp/cicsts/cicsts32/lib/dfjcics.jar
250 Transfer completed successfully.
ftp: 139283 bytes received in 0.02Seconds 6964.15Kbytes/sec.
ftp> quit
221 Quit command received. Goodbye.

C:\Documents and Settings\TOT195\IBM\rational\lsdp6.0\workspace\SG24-5275>
```

Now that dfjcics.jar is copied to our project directory, we must tell Rational Developer for System z to include it on the compiler class path:

1. Right-click the project name, and select **Properties**. Switch to the **Java Build Path** category, and click **Add Jars**, as shown in Figure 4-2 on page 68.

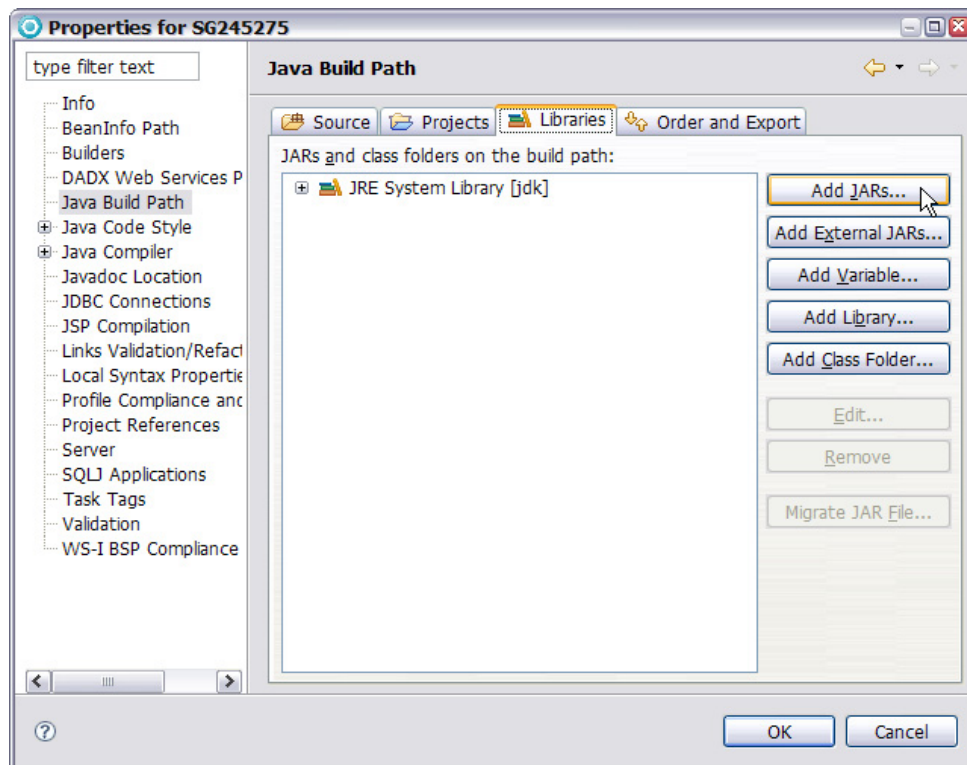


Figure 4-2 Setting up the Java build path in Rational Developer for System z

2. In the dialog that opens, expand the project folder, select **dfjcics.jar**, and then click **OK**. The “JARs and class folders on the build path” list box should now include dfjcics.jar. Click **OK** again to finish the build path setup.
3. Switch to the Source tab in the project properties dialog, select the **Browse** button beside the Default output folder field, and create a new folder named “classes”. Click **OK**, and it will look something like Figure 4-3 on page 69.

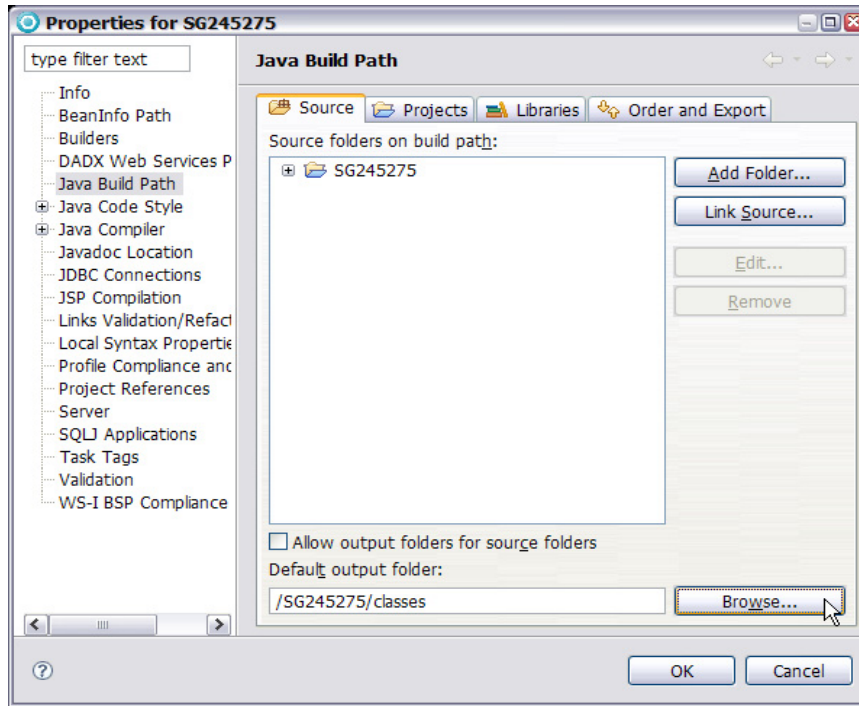


Figure 4-3 Select separate directory for output files.

By doing this, we change the default output directory that contains bytecodes to the classes directory in the project root, which is deployed onto the mainframe later. Click **OK** to close the project properties dialog.

Because the build path changed, Rational Developer for System z rebuilds your project. The compilation errors should be gone because all required libraries are included. In the next step, we deploy the code to the mainframe, and run it in CICS.

Note: If Rational Developer for System z did not rebuild your project after you changed the build path, select **Window** → **Preferences**, select the **Workbench** category, and make sure that Build is checked.

4.2 Deploying and running the program

Now that the code is ready in Rational Developer for System z, the next step is to deploy it to the mainframe, and run it in CICS.

4.2.1 Deploying the code to CICS

You have many choices to move the bytecode to the mainframe. But if you are lucky enough to have Rational Developer for System z installed and configured properly on the mainframe, the deployment is easy and straightforward. We also include steps for the more generic way to use FTP as an alternative.

Exporting using Rational Developer for System z

Before you export the class file, we must make sure that Rational Developer for System z treats the class file as binary files, which means that no conversion is performed during the transfer:

1. Select the **Preference** menu, look for **Remote Systems** → **Files** in the navigator, and click **Add** to add a file type “.class”.
2. Select File Transfer Mode to **Binary**. Click **OK** to apply the changes, and close the dialog.

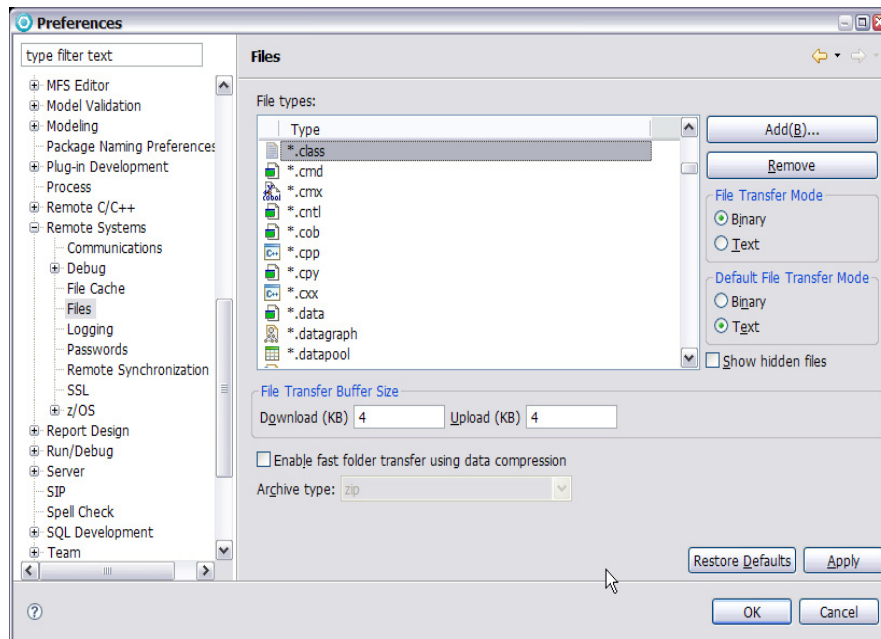


Figure 4-4 Tell Rational Developer for System z to treat class file as binary

Now we can export class files to the mainframe UNIX System Services directory:

1. Select **Export** in the context menu of the project, select **Other** → **Remote file system**, and select **Next**.
2. In the next page of the wizard, select only the classes directory in the project root, and then click **Browse** to select the target directory on your remote system (This directory should be a part of your classpath in the JVM profile used by the CICS region), also select **Create directory structure for files**.
3. Make sure everything looks good, and then click **Finish**. Now your Java class file is on its way to the mainframe.

Exporting using FTP

If your installation does not offer Rational Developer for System z access to the zFS file system, you can use FTP to export your code. Fortunately, you do not have to do this manually; Rational Developer for System z comes with built-in FTP support.

To export the code:

1. Select the project, and then in the context menu, select **Export**.
2. From the list of export destinations, select **Other** → **FTP**, and click **Next**.
3. Click **Browse**, and select the classes directory to upload.
4. In the FTP host field, type the TCP/IP host name of your z/OS machine.

5. In the FTP folder field, type the name of the target zFS directory, in our example, /u/cicsrs6/ (see Figure 4-5). Click **Next**.
6. Enter your user name and password. Click **Finish**.

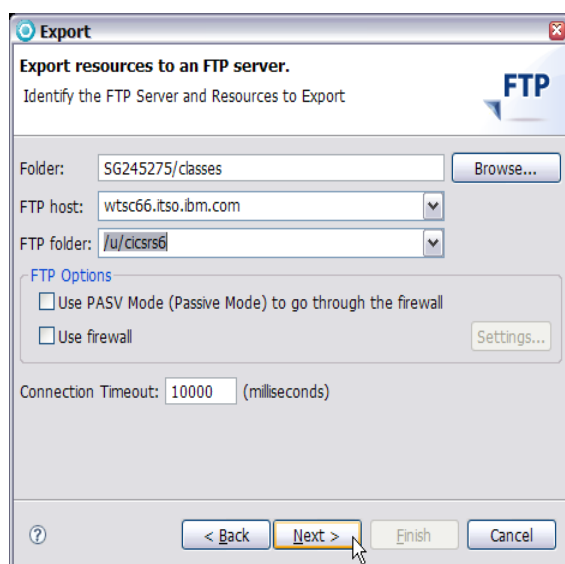


Figure 4-5 Export to an FTP server wizard

Note: Unfortunately, the FTP export does not allow you to do ASCII-to-EBCDIC translation—it always transfers files in binary, which is OK for the .class files, but not if you want to ship the source code to the mainframe.

4.2.2 Setting up the transaction and program definitions

Next, we must set up the CICS transaction and program definitions for our program. We assume that all JVM-related set up, such as creating a JVM profile, is already complete. Also, we assume that a transaction and program definition are created for you to use, and that you are authorized to view and change the program definition attributes.

In the following discussion, we use transaction id HELO and program name HELOWRLD for our example. Depending on your installation standards, your friendly CICS system programmer might have set up different names.

1. First, we verify that the transaction's initial program is set up to be HELOWRLD. Clear the CICS panel, and type `CEMT INQUIRE TRANSACTION(HELO)`. CICS shows a panel that shows a summary of the HELO transaction's attributes:

INQUIRE TRANSACTION(HELO)

STATUS: RESULTS - OVERTYPE TO MODIFY

Tra(HELO)	Pri(001)	Pro(HELOWRLD)	Tcl(DFHTCL00)	Ena Sta
		Prf(DFHCICST)	Uda Bel Iso	Bac Wai

Thus, we verified that the transaction's initial program is indeed HELOWRLD.

2. Next, we look at the definition of program HELOWRLD.

INQUIRE PROGRAM (HELOWRLD)

STATUS: RESULTS - OVERTYPE TO MODIFY

Prog(HELOWRLD)	Jav Pro Ena	Ced
Res(001)	Bel Uex Ful Thr	Jvm

3. Expand the display by tabbing to the first line of the summary, and press **Enter**. Your display should look similar to Figure 4-6; however, we left out several attributes that are not relevant to this discussion.

```
INQUIRE PROGRAM (HELOWRLD)
RESULT - OVERTYPE TO MODIFY
  Program(HELOWRLD)
  Language(Java)
  Progtype(Program)
  Status( Enabled )
  Copystatus( Notrequired )
  Cedfstatus( Cdf )
  Dynamstatus(Notdynamic)
  Rescount(001)
  Usecount()
  Exekey(Uexekey)
  Executionset( Fullapi )
  Concurrency(Threadsafe)
  Remotesystem()
  Runtime( Jvm )
  Jvmclass( com.ibm.itso.sg245275.HelloWorld )
  Jvmprofile( DFHJVMUG )
```

Figure 4-6 Program definition attributes

4. Verify that the Runtime attribute is set to Jvm and that the Jvmclass attribute is set to the Java class name of the program. If this is not the case, you can overwrite the previous values, and press **Enter** to apply the changes. Also, verify that the JVM profile name is correct.

4.2.3 Running the program

Now, we are finally ready to run our program:

1. Clear the CICS panel, and enter the transaction code, as shown in Figure 4-7.

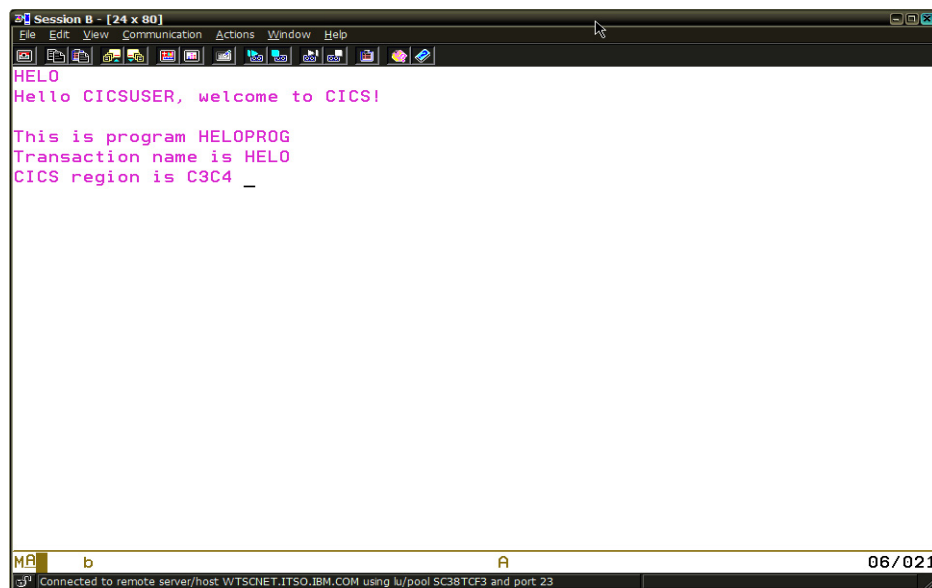


Figure 4-7 Output from the HELO transaction

If this did not run error free, refer to 4.2.4, “Troubleshooting” on page 74. If not, congratulations, you successfully ran your first program in the CICS Transaction Server.

- Next, invoke the transaction in a way such that it is not associated with a terminal (in CICS terms, it does not have a principal facility). If the program works correctly, output goes to a zFS file instead. To do this, we use the CICS-supplied CECI transaction.
- You can use the command-level interpreter (CECI) transaction to check the syntax of CICS commands and to process these commands interactively on a 3270 panel. Using CECI, you can follow through most of the commands to execution and display the results.

Tip: For a full description of CECI, refer to *CICS Supplied Transactions*, SC34-6230.

In 6.12, “Interval control” on page 124, we discuss *Interval control* services, which allows one transaction to asynchronously start another transaction. We now do that manually, using CECI, to start our Hello program:

- On the CICS panel, type CECI, and press **Enter**. You get a list of all CICS commands, as shown in Figure 4-8.

STATUS: ENTER ONE OF THE FOLLOWING					
ABend	DELAy	FREE	POP	RETRieve	SUSpend
ACquire	DELETE	FREEMain	POSt	RETurn	SYncpoint
ADD	DELETEQ	GDs	PURge	REWInd	TEst
ADDRess	DEQ	GET	PUSH	REWRite	TRace
ALlocate	DISAble	GETMain	PUT	ROute	UNlock
ASKtime	DISCard	GETNext	Query	RUn	UPdate
ASSign	DOcument	Handle	READ	SENd	Verify
BIF	DUmp	Ignore	READNext	SET	WAIT
BUild	ENABle	INquire	READPrev	SIGNOff	WAITCics
CANcel	ENDBR	ISsue	READQ	SIGNON	WEb
CHAnge	ENDBROwse	Journal	RECeive	SPOOLCclose	WRITE
CHEck	ENQ	LIInk	RELease	SPOOLOpen	WRITEQ
COLlect	ENTER	LOad	REMOve	SPOOLRead	Xct1
CONNect	EXtract	MONitor	RESET	SPOOLWrite	
CONVerse	FEpi	MOVE	RESETBr	START	
CReate	FORCe	PERform	RESUme	STARTBR	
DEFine	FORMattime	POInt	RESYnc	STARTBROwse	
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER				9 MSG	

Figure 4-8 CECI transaction: Initial panel

- We want to use the START command, which starts another transaction. Type START TR(HELO), and press **Enter**. CECI checks the command syntax, and because it is valid, responds with the message ABOUT TO EXECUTE COMMAND.
- Press **Enter** again to actually execute the command. CECI responds with COMMAND EXECUTION COMPLETE.

This way of invocation causes the transaction to be run without a terminal being associated with it; therefore, the output from our program goes to a zFS file. The name and location of that zFS file are configured in the JVM profile under which the program was set up to run.

Example 4-3 on page 74 shows the zFS file from our example.

Example 4-3 zFS file /u/cicsvr6/work/A6POC3C4/dfhjvmout in our example

Hello CICSUSER, welcome to CICS!

This is program HELOPROG
Transaction name is HELO
CICS region is C3C4

Next, you can experiment with the debugging techniques that we explain in Chapter 8, “Problem determination and debugging” on page 193, for example, you might want to try the execution diagnostic facility (CEDF), or you can run the program under the control of the debugger to see what is going on.

4.2.4 Troubleshooting

If the program failed to run for whatever reason, a message similar to the following is displayed on the bottom of the CICS panel:

```
DFHAC2206 20:56:01 SCSCPJA7 Transaction HELO failed with abend AJ07. Updates  
to local recoverable resources backed out.
```

The most important piece of information is the *abend code*, which indicates what went wrong. (The abend code in itself is not exactly self-explanatory, but you can get an idea of what went wrong.)

Tip: See *CICS Messages and Codes*, GC34-6241, for a detailed explanation of each transaction abend code.

In fact, abend code AJ04 is one that you might see quite often. It means that the class that contains the main method of your application was not found, probably because your classpath is set up incorrectly or you misspelled the main class name on the program, see definition. Review 8.2.1, “Abend AJ04” on page 196, for more on this code.

Another common abend code is AJ05, which means that CICS code caught an unhandled exception, that is, an exception was thrown but never caught by your application, all the way up and beyond your application’s main method.

See Chapter 8, “Problem determination and debugging” on page 193, for more information about problem determination.



Writing Java 5 applications for CICS

In this chapter, we discuss the changes that you might want to consider when migrating your Java application to run on Java 5. We also offer some guidance for Java programmers who are relatively new to CICS and z/OS.

5.1 Migrating Java applications to Java 5

Migrating to Java 5 brings opportunities to use new features of the Java language to write more maintainable and better-performing programs and some small potential challenges. In this section, we help you to deal with some of the problems that might be introduced by migration to Java 5, and then we show you how you can benefit from the advantages offered by the new features.

5.2 New compiler errors and warnings

Migrating existing programs to Java 5 can bring some small challenges, even if you do not use the new features. New compiler warnings are not uncommon due to differences in coding syntax brought about by the new language features. While these warnings tend to be harmless, it is useful to know why they are being issued, and it is worth considering updating code to make use of the new features.

In the following sections, we discuss some of the errors and warnings that you are most likely to see, along with their cause and what action is appropriate (if any).

Of course, you can avoid all of the errors and warnings by compiling with the '-source 1.4' flag, but this does rather defeat the point of switching to the newer version of Java.

5.2.1 Error: syntax error on token 'enum'

Error: syntax error on token 'enum'. This occurs when you have code similar to this:

```
Vector myVector = new Vector();  
Enumeration enum = myVector.elements();
```

The 'enum' is now a keyword in Java, so this error requires a change to the variable name. While you are making that change, consider the use of the Iterator class in preference to Enumeration and even replacing Vector with one of the List implementors, such as ArrayList or LinkedList.

More information about the new keyword 'enum' is in 5.3.4, “Typesafe enums” on page 81.

5.2.2 Warning: ClassX is a raw type. References to generic type ClassX<E> should be parameterized

This warning is issued for any of the collection classes, such as ArrayList, Map, and so on, where 'ClassX' is replaced by the collection class name. The warning is pointing out that Java 5 introduced an extension to the collection classes that allows the type of object to be stored in the collection to be specified and checked at compile time, rather than at runtime.

This is only a warning and does not stop the code compiling, and the code will still run as it did under Java 1.4 without modifications, but much can be gained in terms of code readability, maintenance, reduction in errors, and even performance by making use of this new feature. For more information, review 5.3.1, “Generics” on page 77.

5.2.3 Removed Error for boxing/unboxing

Prior to Java 5, if you tried to use a primitive value where you should use an Object, and vice-versa, then you received an error. Java 5 introduced automatic boxing and unboxing, which automatically converts between primitives and their wrapper classes. Although this avoids compiler errors and simplifies code, it does not remove the runtime overhead of these conversions. Take care to use the appropriate type, for example, objects where the value must be stored in a collection class, and primitives where the value is to be used in a calculation or a boolean operation. For more details, see the section 5.3.3, “Autoboxing and unboxing” on page 80.

5.3 Using the new features in Java 5

After you are running Java 5, you benefit by using the new features that Java 5 offers.

For a complete description of the new features, see the Java 5 documentation. Highlights of the new features include:

- ▶ Generics
- ▶ Typesafe enums
- ▶ Enhanced for loop
- ▶ Autoboxing & unboxing

Keep in mind when considering to use these new features that they do not run on older versions of Java. Any code that you write to use these features only runs on Java 5 or later. Therefore, wait until you migrate your Java runtimes to Java 5 before updating your code to exploit these new features.

5.3.1 Generics

Java has provided a rich set of collection classes since Java 2. These collection classes provide efficient ways to manage large and small collections of objects efficiently, and are the preferred way to handle collections of objects, since their introduction. They also had one significant shortcoming: There was no way to specify the type of object being stored in the collection. All objects were stored as instances of Object, and had to be cast into their actual type when they were extracted from the collection, which resulted in messy code with casts when ever objects were extracted from the collection. Worst still, as any type of object could be added to the collection, it was sometimes necessary to test the type of object before the cast was performed or risk runtime exceptions when a cast was attempted into the wrong class.

Java generics provide a simple solution to these problems by offering type-safe versions of the collection classes.

Let us start with a simple piece of example code written without the use of generics. This simple piece of code in Example 5-1 counts the number of uses of the word 'CICS' in the passed collection.

Example 5-1

```
public static int countCICS(Collection c) {  
    int count = 0;  
    Iterator iterator = c.iterator();  
    while (iterator.hasNext()) {  
        if (((String) iterator.next()).equals("CICS"))
```

```
        count++;
    }
    return count;
}
```

With this code we would want to state in the documentation of the method that the passed collection only contains instances of the String class and hope that who ever uses this method reads the comment and obeys it.

Example 5-2 is a simple piece of code that invokes that method.

Example 5-2

```
public static void testCountCICS() {
    List al = new ArrayList();
    al.add("Hello");
    al.add("CICS");
    al.add("Programmer");
    al.add(new Integer(4));           // (1)

    int count = countCICS(al);
}
```

While the code in Example 5-2 compiles without errors, when we try to run the program we get a `ClassCastException` because of the addition of an Integer object at line (1) when the `countCICS()` method attempts to cast this Integer into a String.

We also have the same methods implemented using generics. Example 5-3 is the updated version of `countCICS()`.

Example 5-3

```
public static int countCICS(Collection<String> c) {
    int count = 0;
    Iterator<String> iterator = c.iterator();
    while (iterator.hasNext()) {
        if (iterator.next().equals("CICS"))
            count++;
    }
    return count;
}
```

The changed code is highlighted. The addition of '`<String>`' to the declaration of the Collection and Iterator state that these only hold String objects. Because the Iterator declared that it only handles String objects, there is no longer a need for a cast when the values are read from the iterator in the 'if' statement.

The code '`<Type>`' can be read as 'of Type'.

Example 5-4 is an updated version of the test code, but this time using generics when we create the collection to be processed.

Example 5-4

```
public static void testCountCICS() {
    List<String> al = new ArrayList<String>();
    al.add("Hello");
}
```



```

    al.add("CICS");
    al.add("Programmer");
    al.add(new Integer(4));           // (1)

    int count = countCICS(al);
}

```

Once again, the changed code is highlighted. The declaration of the List and the creation of the ArrayList also have '<String>' appended to them to state that they only accept String objects.

This code will fail with a compile error on the line labeled (1) where it tries to add the Integer to the collection. With the use of generics, the compiler enforces the requirement for the collection to only contain Strings, which reduces the risk of errors at runtime. Removing or correcting this line removes the compile time error, and we can then have confidence that the code does not fail at runtime. If the code compiles without warnings, then we know that we will not have a ClassCastException at runtime. We also have the benefit of simpler code without the cast and less parentheses in the 'if' statement.

Generics are compatible with the existing collection classes, which allows easier migration of existing code. So, we can invoke the new version of countCICS() (Example 5-3 on page 78) with the old version of the test code (Example 5-2 on page 78). Of course, this still gives us a runtime error because of the attempted cast of the Integer object into a String. After this problem is resolved we still have a warning message telling us that the conversion from old (un-typed) collection to a new (typed) collection cannot be guaranteed to be type safe because we do not know at compile time whether the collection only contains objects of the correct type.

Correspondingly, the new test method (Example 5-4 on page 78) can be used to invoke the old countCICS() method (Example 5-1 on page 77). After the invalid line at (1) is corrected or removed, this code compiles and runs without errors.

Programmers who are familiar with C++ might notice that the syntax of Java generics is similar to C++ templates, but note that while they might look similar, the implementation is very different and the two should not be confused.

5.3.2 Enhanced for loop

Java offers the typical for loop construct where of the form

```
for (initialiser; exit condition; incrementor)
```

With Java 5, the 'For-Each' loop is introduced, which has syntax of the form:

```
for (element : collection)
```

The colon (:) can be read as 'in'. This version of the for loop can be thought of as 'for each item in the collection'. The syntax, and its advantages, are probably better demonstrated using an example, Example 5-5, such another re-work of the countCICS() method as used in the previous examples.

Example 5-5

```

public static int countCICS(Collection<String> c) {
    int count = 0;
    for (String s: c) {
        if (s.equals("CICS"))

```

```
        count++;
    }
    return count;
}
```

Example 5-5 on page 79 also demonstrates the new form of the 'for' loop. The complexity of the code is reduced further. When compiled, the code gains an implicit Iterator, but this is hidden from the programmer, which results in simple and easy-to-read code.

There are limitations to this code, for example, if we wanted to do more than merely count the instances of the word 'CICS', such as remove all instances of the word 'CICS', then we want to invoke the remove() method on the Iterator. Because we do not have access to the Iterator, we cannot do this, and so this version of the for loop is not appropriate.

There is not much to be gained from replacing existing code with the new for loop syntax, but it is a useful construct to consider using in new code.

This code also demonstrates the questionable practice of using single character variable names, which of course is not recommended in production code.

5.3.3 Autoboxing and unboxing

Because the Java collection classes only hold Objects and not primitives, it is sometimes necessary to 'box' primitives in their appropriate wrapper class to store them in a collection and then 'unbox' them to get the primitive value back when the object is read from the collection. Although this code might be trivial to write, it does add clutter to the code.

A slightly contrived example of boxing and unboxing might be something like the code that creates an array of Strings, calls a method that returns the passed Strings mapped to the length of each String, and then reports on the lengths of the Strings, as shown in Example 5-6. The first version of the code uses explicit boxing and unboxing.

Example 5-6

```
public static void testStringLengths() {
    String[] strings = new String[] {"Hello", "CICS", "Programmer"};
    Map<String, Integer> result = getStringLengths(strings);
    for (Map.Entry<String, Integer> e: result.entrySet())
        System.out.println("String " + e.getKey() + " length is " +
            e.getValue().intValue());
}

public static Map<String, Integer> getStringLengths(String[] strings) {
    Map<String, Integer> map = new HashMap<String, Integer>();
    for (String s : strings) {
        map.put(s, new Integer(s.length()));
    }
    return map;
}
```

The code also provides another demonstration of the use of generics and the enhanced for loop.

Now, the same code with automatic boxing and unboxing looks like Example 5-7 on page 81.

Example 5-7

```
public static void testStringLengths() {
    String[] strings = new String[] { "Hello", "CICS", "Programmer" };
    Map<String, Integer> result = getStringLengths(strings);
    for (Map.Entry<String, Integer> e: result.entrySet())
        System.out.println("String " + e.getKey() + " length is " + e.getValue());
}

public static Map<String, Integer> getStringLengths(String[] strings) {
    Map<String, Integer> map = new HashMap<String, Integer>();
    for (String s : strings) {
        map.put(s, s.length());
    }
    return map;
}
```

The changed lines are in bold text.

Boxing and unboxing, whether automatic or manual, has a performance overhead. Boxing creates a new object, which not only takes some processing, it also results in another object (the wrapper instance) on the heap that must be garbage collected at some time. Unboxing requires a method call to read the primitive value from the wrapper class. Because of this, only use the boxing and the wrapper classes when they are required to avoid unnecessary performance impacts. As the introduction of automatic boxing and unboxing hides the process, take additional care to avoid unnecessary conversion and the associated impact on performance.

5.3.4 Typesafe enums

A common approach to enumerated values in Java is to use static int values like Example 5-8.

Example 5-8

```
public static final int CONNECTION_OPEN = 0;
public static final int CONNECTION_CLOSING = 1;
public static final int CONNECTION_CLOSED = 2;

public static final int SECURITY_SSL = 0;
public static final int SECURITY_NO_SSL = 1;

public void createConnection(int initialState, int securitySetting) { ...}

public void someMethod() {
    createConnection(OPEN, NO_SSL) // (1)
    createConnection(NO_SSL, OPEN) // (2)
}
```

Although the code in Example 5-8 will compile, it has many problems, which include:

- ▶ It is not typesafe: While lines (1) and (2) both compile, one of them has unexpected results.
- ▶ Trace values are unhelpful: Seeing the int values in the trace tells you very little without reading the program source code.

- The variable names must include their context because there is no namespace that is associated with the int variables.
- It is brittle: If a new value is added in the middle of the values or the numbering is changed, any code using the values must be re-compiled.

It is possible to write code that addresses these issues, such as Example 5-9.

Example 5-9

```

public class ConnectionStatus {
    public static final ConnectionStatus OPEN = new ConnectionStatus("Open", 1);
    public static final ConnectionStatus CLOSING = new ConnectionStatus("Closing",
2);
    public static final ConnectionStatus CLOSED = new ConnectionStatus("Closed",
3);

    private final int value;
    private final String description;

    private ConnectionStatus(String description, int value) {
        this.description=description;
        this.value = value;
    }

    public int getValue() {return value; }
    public String getDescription() {return description;}
    public String toString() { return description + ":" + value; }
}

public class Security {
    public static final Security SSL = new Security("SSL", 1);
    public static final Security NO_SSL = new Security("No SSL", 2);

    private final int value;
    private final String description;

    private Security(String description, int value) {
        this.description=description;
        this.value = value;
    }

    public int getValue() {return value; }
    public String getDescription() {return description;}
    public String toString() { return description + ":" + value; }
}

public void createConnection(ConnectionStatus initialState, Security
securitySetting) {...}

public void someMethod() {
    createConnection(ConnectionStatus.OPEN, Security.NO_SSL);
}

```

Example 5-9 on page 82 addresses the issues, but it is verbose. We do have the benefit that if the `initialState` and `securitySetting` parameters are reversed, we get a compile error that is easy to fix.

With the introduction of the `'enum'` keyword, we can have the advantages of the dedicated class, but without so much typing. Example 5-9 on page 82 can be coded in a smaller form, as shown in Example 5-10.

Example 5-10

```
public enum ConnectionStatus {OPEN, CLOSING, CLOSED};
public enum Security {SSL, NO_SSL};

public void createConnection(ConnectionStatus initialState, Security
securitySetting) {...}

public void someMethod() {
    createConnection(ConnectionStatus.OPEN, Security.NO_SSL);
}
```

The enum values can also be used as a type for generics and even can be used in the enhanced for loop in a line, such as:

```
for (ConnectionStatus state : ConnectionStatus.values()) { ... }
```

There are other advantages to the new enum types, including the ability to add enum values without breaking existing code, extensions to the Java Collections classes, and add methods or interface implementations to an enum.

5.4 Introduction to CICS for Java programmers

There are differences between writing Java programs for CICS and writing Java programs for other environments. In this section, we briefly discuss some of these issues, but this is not intended to be a complete description of CICS. For a complete description, review the CICS Transaction Server documentation.

CICS is a transaction processing system, which means that it provides services for a user to run applications online at the same time as many other users are submitting requests to run the same or other applications, using the same files and programs. CICS manages the sharing of resources, integrity of data, and prioritization of execution, while maintaining fast response times.

A CICS application is a collection of related programs that together perform a business operation, such as processing a product order or preparing a company payroll. CICS applications execute under CICS control using CICS services and interfaces to access programs and files.

You can write CICS application programs in a variety of languages, including COBOL, Assembler, C, C++, PL/I, and of course, Java. Most of these languages access CICS services through the `'EXEC CICS'` interface. Java, however, does not use the `EXEC CICS` interface and must use the `JCICS` classes to access CICS services.

CICS provides a selection of services to the application program. For a thorough overview of these services, refer to the CICS Transaction Server documentation; however, here we do provide a summary of some of the key features that are provided to Java programs:

- ▶ Data management services:
 - CICS provides the ability to access VSAM files with data backout or forward recovery in the event of failures.
 - CICS supports access to databases, such as DB2 through appropriate APIs.
 - CICS also provides two proprietary file structures: Temporary Storage (TS) and Transient Data(TD).
- ▶ Communication Services:
 - CICS provides access to a variety of display devices over SNA and TCP/IP protocols.
 - CICS supports communication inbound and outbound over IIOP using an Object Request broker (ORB).
 - CICS also provides a selection of proprietary protocols that allow programs to access programs and other resources in other regions and asynchronously start transactions in the same or other CICS regions.
 - Requests to run programs that are defined in other CICS regions can be automatically routed to the other region with the results being routed automatically back to the calling region.
- ▶ Unit of work services:
 - CICS supports ACID transactions and gives programs the ability to programmatically commit or rollback transactions.

These features are available to Java programs through the JCICS classes, which we discuss in 6.1, “Introduction to JCICS” on page 90. There are also other services that are provided in CICS that are not exposed to CICS Java programs.

5.5 CICS program design guidelines

Designs for programs that run in CICS, or in other multi-user environments, can be different to designing programs that run in single-user environments. The issues and guidance for designing such applications are worthy of a book in themselves and are outside the scope of this book; however, some basic principles are worth stating here for you to keep in mind when you design and write CICS Java programs to help you along the road to writing efficient and well-behaved Java programs for CICS:

- ▶ CICS systems handle multiple transactions and multiple users simultaneously.
- ▶ CICS applications frequently are distributed across multiple CICS regions with the work frequently being routed dynamically to a system with available resources.
- ▶ CICS programs typically must be highly scalable, which means that resources must be shared between multiple programs and often multiple instances of the same program, often in different CICS regions.
- ▶ The program must be written in such a way as to avoid multiple users updating the same record at the same time, which CICS services offer help with.

- ▶ The program must be designed to have a clean approach to error handling:
 - It is essential that when an error occurs, sufficient information is provided to allow successful debugging of the program. Empty 'catch' blocks fail to do this and endear their author to the programmer attempting to debug the problem.
 - Any resources that the program holds must be released both when the program ends normally and in the case of an error.
- ▶ Programs must be designed to not make excessive demands on system resources, such as processor time, memory, communication with auxiliary storage, and communication bandwidth. Java programmers should pay particular attention to avoid creating excessive and unnecessary objects on the heap.
- ▶ Ensure that private data from one program invocation is not available to other invocations of programs, which requires particular care when writing Java programs that are executed in a continuous JVM.
- ▶ CICS programs typically exist and are modified over a long period of time. Programs must be written in such a way to be easily maintained and modified by other programmers in the future.

Because of these issues, Java programmers must take care to write well-behaved Java applications. Programs that hog resources, perform badly, or create large numbers of unnecessary objects in the Java heap can severely impact the performance of the CICS systems. Even a properly configured and carefully turned set of CICS regions will not perform well if the application is badly written.

5.6 Differences in Java with CICS

A CICS Java program should be written to work through CICS when accessing system resources to ensure that CICS can manage those resources effectively and ensure that they are shared between all of the programs that are running in the system. CICS JVMs are typically re-used, so take care to not leave resources 'lying around' between uses of the JVM because this impacts transaction isolation and can impact performance and resource usage.

Because of this, there are certain Java language features and APIs that should not be used by Java applications that are running in CICS, and others should be used with care.

Java resources that should either not be used or that only should be used with caution are discussed in the next sections and alternatives are suggested where appropriate.

There are also third-party APIs, drivers, and extensions that can be used in Java programs. Many of these can be used in Java in CICS, but be aware that they typically do not benefit from CICS services, such as transaction support.

5.6.1 Threads

Java programs can and often do start new threads, but avoid this when the Java programs are running in CICS. Only the thread that invoked the public static main method (the initial program thread - IPT) has access to CICS resources. Additionally, any thread that the IPT starts is not automatically destroyed when the IPT terminates, which damages isolation of the transaction running in the JVM and has the potential to waste system resources.

Because CICS provides the ability for large numbers of programs to run simultaneously and manages access to resources and communication between the programs, it typically should not be necessary for a Java program in CICS to start new threads.

5.6.2 Sockets

Sockets that are created using the classes in package `java.net` use the socket support of the JVM. CICS cannot transactionally manage these sockets, and if they are created by threads that the Java program creates, they can outlive the program that created them and compromise the isolation of the transaction running in the JVM, which has the potential to waste system resources.

CICS system programmers are encouraged to use the Java 2 security policy to stop the creation of the Java sockets, so any program that is written to use these sockets might not run in a production system.

JCICS provides support for TCP/IP. HTTP and web services through CICS.

5.6.3 File I/O

The JCICS API provides access to VSAM files through CICS services and should be used in preference to the `java.io.File` class and related classes. If access is required to a PDS, the JZOS libraries should provide a suitable API.

Modifications to hfs, zFS(files), or PDS members cannot be backed out by CICS if the transaction is rolled back, so take care if you apply updates to these files.

5.6.4 Static data

Static storage is not re-initialized for each invocation of the program in a continuous JVM. So take special care with static fields in Java programs for CICS. This can be both a blessing, and a challenge.

Data that is likely to be needed repeatedly in the JVM can be initialized in the static values and re-used by future invocations of the program in that JVM. Do not assume that the static values were already set because the different invocations of a program might run on different JVMs, and other programs might also run in the same JVM. This re-use of static value can, when used properly, improve program performance.

Do not store data that must not be shared between invocations of a program, such as customer account numbers, in static fields, or the fields should be initialized before the program terminates; otherwise, transaction isolation is impacted.

The CICS Java documentation offers the following guidelines:

- ▶ Define a class field as private and final, whenever possible. Be aware that a native method can write to a final class field, and a non-private method can obtain the object referenced by the class field and can change the state of the object or array.
- ▶ Be aware of system-loaded classes that use changeable class fields.

The 'CICS JVM Application Isolation Utility' tool that is provided with CICS can help you to check your usage of static fields and classes. We describe this tool in 2.4, "Analyzing programs for use in a continuous JVM" on page 23.

Further information is in the CICS Java documentation.

5.6.5 Modifying the JVM state

Although it is occasionally desirable to modify the state of the JVM, such as, changing the default time zone, programs must restore the value to its original setting before the program completes so that the changes do not affect programs that run subsequently in the JVM.

5.6.6 Releasing resources at the end of program execution

It is essential to ensure that resources that a program uses are released at the end of the program, whether the program ends normally or in an error condition. Failure to do so can waste system resources, cause problems with resources being locked out from other programs, and in the example of DB2 connections, stop a new connection being opened by subsequent programs that are running in the JVM due to the existence of a connection that was already opened and not closed by a previous program.

5.6.7 Object Request Broker (ORB)

Java programs in CICS that use the Object Request Broker (ORB) use the CICS ORB. This ORB implementation uses CICS services. For details of support provided by this ORB, see the CICS documentation.

5.7 Data type conversion

CICS runs on System z machines and uses the Extended Binary Coded Decimal Interchange Code (EBCDIC) character set, rather than ASCII, which is the case with many other machines. While Java programs use unicode for its internal representation of characters and strings, conversion to and from the native character set of the host machine is often required. Do not assume that the host machine's character set is ASCII or unexpected errors will occur.

5.7.1 ASCII & EBCDIC issues

There are several common problems with code page conversions in Java and other languages when working across platforms. Java provides help with many of these problems, and you can avoid them with careful coding.

One common problem is assumptions about the line separator character. It is common to write code that expresses the line separator character as '\n'. This only works on some platforms and is a common cause of problems. Even using the `println(...)` method can be a problem. To avoid problems with line separators, it is safest to use code that works on all platforms, such as:

```
String lineBreak = System.getProperty("line.separator");
out.print("Line of text.");
out.print(lineBreak);
```

When you read or write text, you typically want to use the default encoding of the platform that is running the code, such as, EBCDIC when running in CICS, but if the text came from or is being sent to another platform, a different encoding might be required.

XML can offer particular problems. XML documents are normally stored as text, but they store the encoding name within them. So an XML document might begin:

```
<?xml version="1.0"? encoding="UTF-8">
```

An XML document from another platform is typically transferred to the host machine that is running the CICS region as a text file, so the text that makes up the contents of the file is converted into EBCDIC, and this of course conflicts with the encoding, as specified in the header. Alternatively, the file can be transferred in binary mode, in which case the encoding tag is correct, but the file is difficult to read as a text file because it is not encoded in EBCDIC. Similar problems exist with XML that is generated on the host and transferred to a non-EBCDIC machine.

5.7.2 Conversion to and from COBOL, PL/I, and Assembler data types

It is often necessary for CICS Java programs to interact with existing COBOL, PL/I, or assembler programs in CICS. These languages support data types that do not exist natively in Java, such as packed decimal and zoned decimal numbers.

Usefully, this conversion is discussed in the existing IBM Redbooks publication *Java Stand-alone Applications on z/OS Volume 1*, SG24-7177, and code is provided in the additional material with that book that carries out these conversions. In these cases, it is necessary to specify the code page to be used in the conversion. Java provides the ability to do this and the character set for conversions can be specified on classes such as `InputStreamReader` and `OutputStreamWriter` in the 'java.io' package.



The Java CICS API

In this chapter, we give you a gentle introduction to Java programming in a CICS environment.

First, we discuss the difference between traditional CICS programming and using CICS services from Java. Then, we briefly introduce the main functional areas of the API with an in-depth discussion of each. Also, we show and explain some example code to help you get started.

6.1 Introduction to JCICS

There is one fundamental difference between using the CICS API from traditional languages, such as COBOL, PL/I, and C, and using it from Java applications. In languages other than Java, CICS commands are implemented as a language extension, that is, they are embedded into the source program bracketed by EXEC CICS...END-EXEC clauses. Therefore, the source program cannot be compiled as it is; instead, you must first run it through the *CICS translator*, which extracts all CICS calls and replaces them by calls into the *CICS language interface module*, which produces a program that can be compiled by the respective language compiler. If you are familiar with database access using embedded SQL, you can see that this is pretty much the same concept.

Figure 6-1 shows the program preparation process for CICS applications by example of a COBOL program.

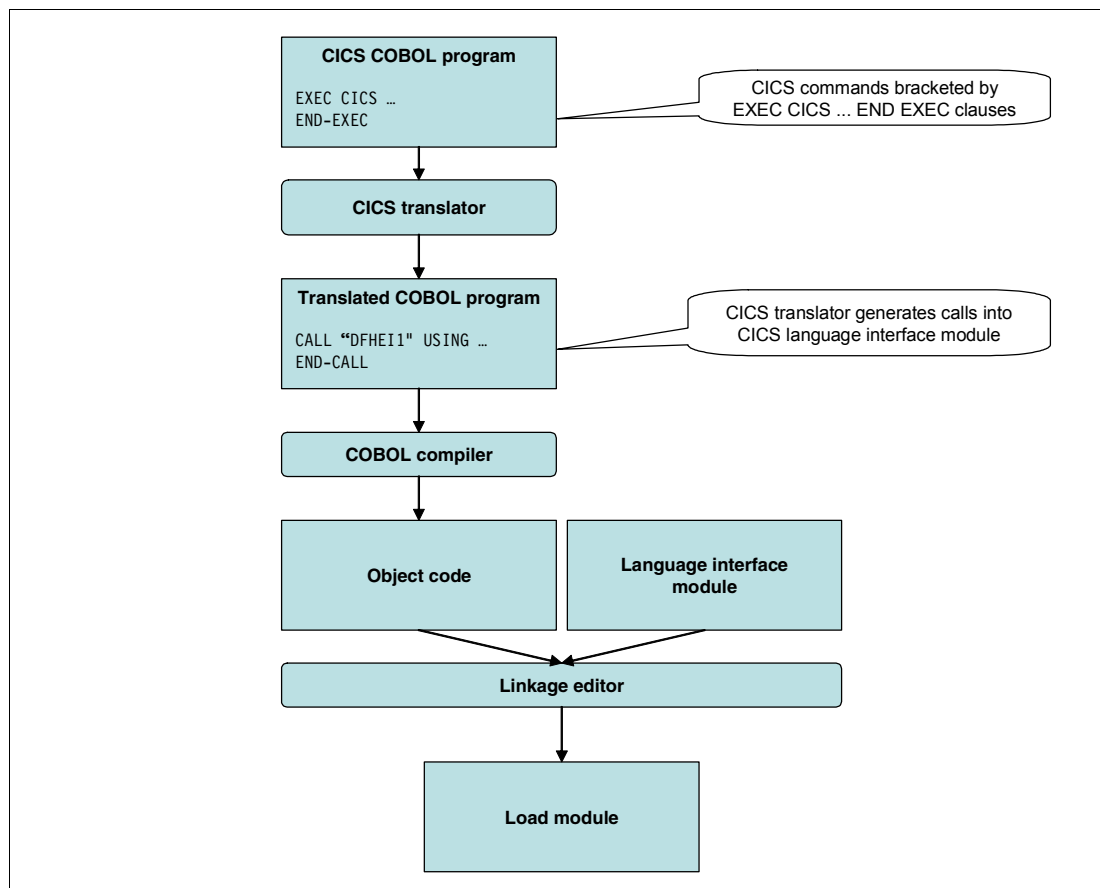


Figure 6-1 Preparing CICS programs in COBOL

In Java, however, things are much simpler. CICS commands are not implemented as a language extension but rather as a regular Java API (the *JCICS* API); therefore, no additional translation steps are necessary. In that sense, CICS programming in Java is no more difficult than using any other Java API.

6.2 A short overview of the JCICS API

In this section, we provide a high-level overview of the CICS API, briefly describing its various categories. Later in this chapter, we describe most of the API in more detail.

Figure 6-2 shows a class diagram of the JCICS API. Due to limited space, we only show some particularly important classes and show more detail in later sections.

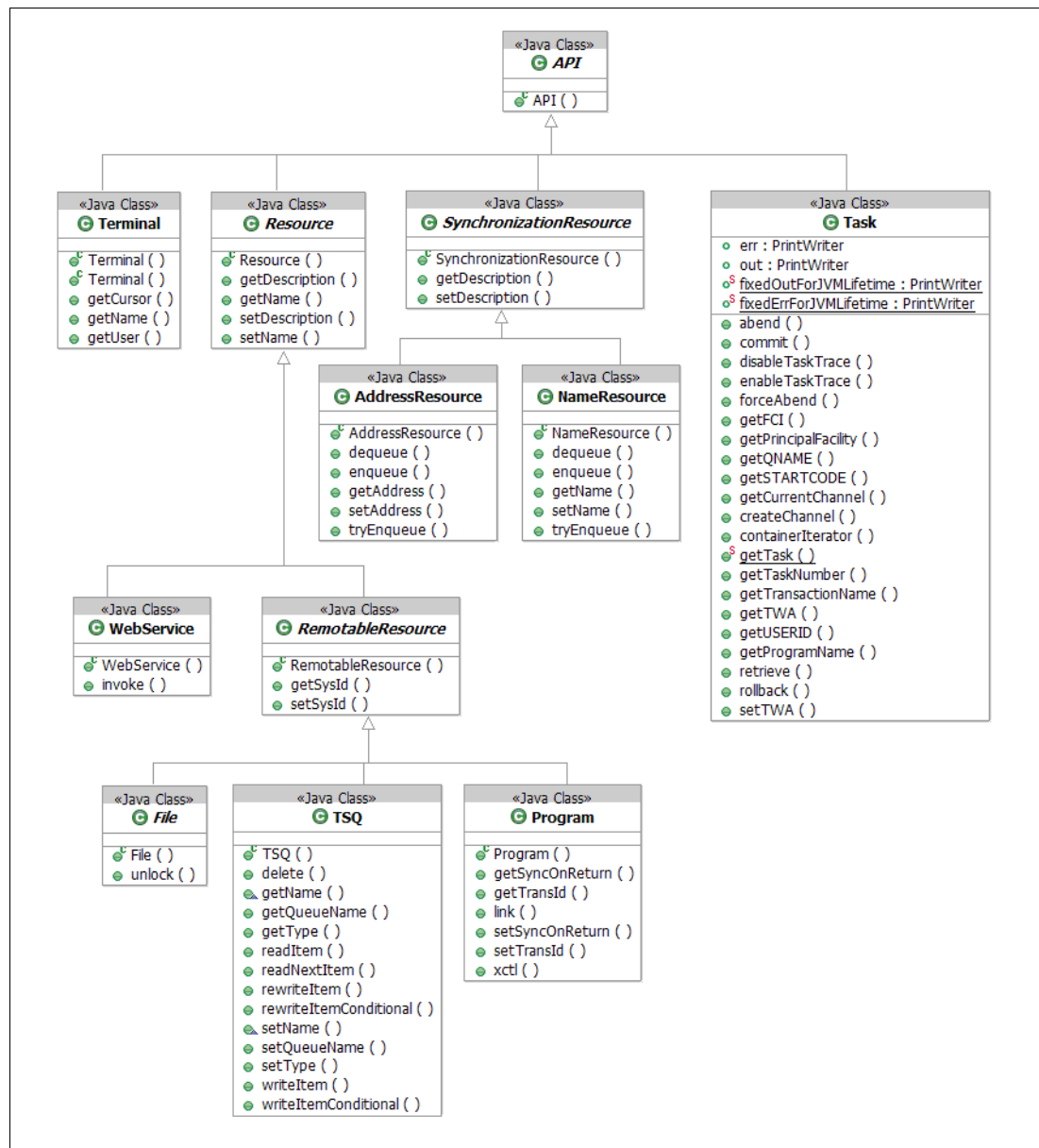


Figure 6-2 Overview of the JCICS API

In the following sections, we briefly describe several functional areas of the API.

6.2.1 Program control

Program control services allow one program to call another program in the same unit of work. Data can be passed to and received back from the called program. Until recently this was

achieved by means of a shared area of storage known as the COMMAREA, but in CICS TS Version 3.1, an additional mechanism known as channels and containers is available as an alternative to a COMMAREA.

6.2.2 File control

Using file control services you can access VSAM files, which come in three different flavors:

- ▶ Key Sequenced Data Sets (KSDS)
- ▶ Entry Sequenced Data Sets (ESDS)
- ▶ Relative Record Data Sets (RRDS)

6.2.3 Synchronization

Synchronization services provide a mechanism to ensure mutual exclusion when dealing with shared resources that must not be modified (or even accessed) by more than one transaction at a time.

6.2.4 Scheduling services

Scheduling services allow a program to start other transactions asynchronously, either immediately or at some specified later time.

6.2.5 Unit of work

This is one of the easiest and probably most important JCICS services to use. You use these services to commit or roll back the current unit of work, which either makes the results of the current transaction permanent or backs them out.

There is no API to explicitly start a new unit of work. A new unit of work is considered active when the previous one ended.

To commit the current unit of work, use the `Task.commit()` method:

```
Task.getTask().commit();
```

To back out the current unit of work, use the `Task.rollback()` method:

```
Task.getTask().rollback();
```

6.2.6 Document services

Document services are most often used in conjunction with CICS Web support services (6.2.7, “Web and TCP/IP services” on page 93) to dynamically build HTML pages that are to be sent back to a Web browser.

You can work with document templates, which are portions of a document that are created offline (or by another program) and might contain symbols that represent dynamic content. You can then replace those symbols to produce the actual document to be sent back.

We show an example of using the Document API in Chapter 7, “Evolving a heritage application using Java” on page 135.

6.2.7 Web and TCP/IP services

Web services allow CICS programs to produce dynamic Web content, probably in response to a client request from a Web browser. In other words, they can act much like CGI programs that are supported by most Web servers or like Java servlets.

The JCICS API provides the following Web-related and TCP/IP-related services:

- ▶ Examining an HTTP request
- ▶ Sending a response back to the client
- ▶ Getting the client's host name
- ▶ Security-related services, such as getting authorization method and certificates

We show how to use Web services in conjunction with Document services in Chapter 7, "Evolving a heritage application using Java" on page 135.

6.2.8 Transient storage queues

Temporary storage is the primary CICS facility for storing data that must be available to multiple transactions. Data items in temporary storage are kept in queues whose names are assigned dynamically by the program that stores the data. Think of a temporary storage queue that contains multiple items as a small data set whose records are addressed either sequentially or directly, by item number. If a queue contains only a single item, think of it as a named scratch-pad area.

6.2.9 Transient data queues

Transient data queues (TD queues) are similar in some respects to transient storage queues, but there are some major differences:

- ▶ They cannot be created on-the-fly but must be predefined.
- ▶ Items can only be read sequentially and cannot be changed.
- ▶ Reading from TD queues is destructive, that is, each item can be read only once. After a transaction reads an item, that item is removed from the queue and is not available to any other transaction.

Today, TD queues are pretty much obsolete, because the preferred alternative is to use WebSphere MQ message queues. Therefore, we do not further discuss TD queues in this book. However, they are fully supported by the JCICS API.

6.2.10 Terminal control

Terminal control services allow an application program to interact with the user's terminal by displaying data on and retrieving user input from the terminal.

However, terminal control in Java is not widely used because there is no JCICS support for an important feature called *Basic Mapping Support* (BMS). To put it simply, BMS is a set of CICS services and tools to create, display, and interact with panel forms from an application program. As you can imagine, BMS panels are much easier to use (and more portable) than direct interaction with the raw terminal.

During the course of this residency, we developed a small package called *JBMS* (for Java Basic Mapping Support), which allows you to create and interact with panel forms, just like the real BMS does. Read all about it in 6.13, "Terminal services" on page 125.

6.2.11 Miscellaneous services

In this section, we cover several services that do not fall into one of the categories above.

Specifically, you can:

- ▶ Inquire about the system name and VTAM® application ID of the CICS region using `Region.getSysId()` and `Region.getAPPLID()`, respectively.
- ▶ Enable and disable tracing in the CICS region: `Region.enableTrace()`, `Region.disableTrace()`.
- ▶ Examine and modify the contents of the *Common Work Area* (CWA), using `Region.getCWA()` and `Region.setCWA()`.
- ▶ Retrieve the transaction name and program name under which the current program is executing, which is useful for logging purposes.
- ▶ Get the name of the user who started the transaction.
- ▶ Examine and modify the *Transaction Work Area* (TWA). The TWA is a small (up to 32 K) area of storage that is allocated when a transaction is initiated and is initialized to binary zeros. It lasts for the entire duration of the transaction and is accessible to all local programs in the transaction. Using the TWA is no longer recommended for new applications.
- ▶ Find out how the current program was started using `Task.getTask().getSTARTCODE()`. Table 6-1 lists the different possible startcodes and their meanings.

Table 6-1 Startcodes as returned by `Task.getTask().getSTARTCODE()`

Startcode	Meaning
D	The task was initiated to process a distributed programming link (DPL) command that did not specify the SYNCONRETURN option. (The task is not allowed to issue syncpoints.)
DS	The task was initiated to process a distributed programming link (DPL) command containing the SYNCONRETURN option. (The task is allowed to issue syncpoints.)
QD	CICS initiated the task to process a transient data queue that reached trigger level.
S	Another task initiated this one using a START command that did not pass data in the FROM option.
SD	Another task initiated this one using a START command that passed data in the FROM option.
SZ	The task was initiated with a FEPI START command (see <i>CICS Front End Programming Interface User's Guide</i> , SC34-6234, for further information).
TO	The task was initiated to process unsolicited input from a terminal (or another system), and the transaction to be executed was determined from the input.
TP	The task was initiated to process unsolicited input or in response to a RETURN IMMEDIATE command in another task. In either case, the transaction to be executed was preset (in the RETURN command or in the associated TERMINAL definition) without reference to input.
U	CICS created the task internally.

6.2.12 Services that the JCICS API does not support

JCICS does not support these CICS services:

- ▶ APPC unmapped conversations
- ▶ CICS Business Transaction Services
- ▶ DUMP services
- ▶ Journal services
- ▶ Storage services
- ▶ Timer services
- ▶ BMS services SEND MAP and RECEIVE MAP

6.3 JCICS basics

Using JCICS application programming classes, you can invoke CICS services from Java. To invoke a CICS service, simply call a Java method. The methods that JCICS provides use Java exception classes to return error messages to the caller. The exception handling mechanism is internally based on the CICS API response codes, which are set after execution of the CICS commands.

CICS resources, such as files, programs, temporary storage queues, and transient data queues, are represented by instances of the appropriate Java class. Therefore, to work with, for example, a temporary storage queue (TSQ), you first create an instance of a TSQ object. Then assign it a name that corresponds to the name of the TSQ that you want to use, and use its methods to manipulate the queue:

```
TSQ tsq = new TSQ();
tsq.setName("MYTSQ");
tsq.writeItem("Hello World!".getBytes());
```

Often, you might want to initialize the JCICS objects that your program works within a constructor, and use the initialized objects from regular methods. In Example 6-1 (an excerpt from 6.8, “Using transient storage queues” on page 109), we declare a TSQ to be a *blank final* instance variable. The *final* keyword indicates that the variable must be initialized in the constructor and cannot be reassigned after it is initialized.

Example 6-1 Declaring a TSQ to be a blank final instance variable

```
public class Scratchpad {

    public final TSQ tsq;

    public Scratchpad(String tsqName) {
        this.tsq = new TSQ();
        tsq.setName(tsqName);
    }

    public void writeBytes(byte[] bytes) throws ItemErrorException, ... {
        try {
            tsq.rewriteItem(1, bytes);
        } catch (InvalidQueueIdException e) {
            tsq.writeItem(bytes);
        }
    }
    ...
}
```

The JCICS library structure and naming conventions are modelled to be fairly close to the traditional EXEC CICS programming interface, so translating the CICS calls in a traditional CICS program to JCICS calls in a Java application is fairly easy. The downside is that the JCICS way of doing things often does not quite “feel right” to a seasoned Java programmer, for example, to iterate over a set of related resources, CICS has the concept of browsing, which is similar to the Java idiom of using an *Iterator*; however, the relevant JCICS classes do not implement the *Iterator* interface; instead, they stand entirely on their own.

Note: All JCICS classes that represent CICS resources are designed to comply with the JavaBeans standard, so you can use them with a visual composition editor.

6.4 Input and output streams

Similar to any standalone Java application, JCICS applications can use the predefined streams: `System.in` to read input and `System.out` and `System.err` to print output.

Additionally, the `Task` class has two fields, `out` and `err`, which are directed at the user’s terminal if the transaction is started from a terminal (in CICS parlance, if the *principal facility* associated with the transaction is a terminal). Otherwise, they are the same as `System.out` and `System.err`, respectively.

However, the standard input stream (`System.in`) is never connected to the terminal. Rather, it is associated with an HFS or zFS file whose name is set up in the JVM profile. In other words, it is not *interactive* in the sense that you can read user input from it. So, there is probably little use for the `System.in` stream, except maybe for reading configuration information at program startup.

`System.out` and `System.err` are connected to HFS or zFS files as well. Those files are created (or scratched) when the JVM starts up. For more information, see 8.3.4, “JVM stdout and stderr”.

The terminal might change each time that the JVM is reused, although `System.in`, `System.out`, and `System.err` remain connected to their respective HFS or zFS files.

6.5 Exception handling

Because anything that can go wrong eventually does go wrong, you must handle error conditions in your code.

As usual, error reporting and handling in JCICS is integrated into the standard Java exception handling mechanism.

In traditional languages, CICS indicates the success or failure of a CICS command by returning a condition code to your application program (sometimes called the RESP value because you use the RESP keyword to retrieve it). If everything went fine, the RESP value is NORMAL. If the RESP value is some value other than NORMAL, you can test the value and obtain what happened. Many RESP values also have an associated RESP2 value, which gives further detail about the error.

In Java, RESP codes are mapped to Java exception classes. For each RESP value that can occur in CICS, there is one corresponding Java exception class.

Figure 6-3 on page 97 shows part of the JCICS exception hierarchy.

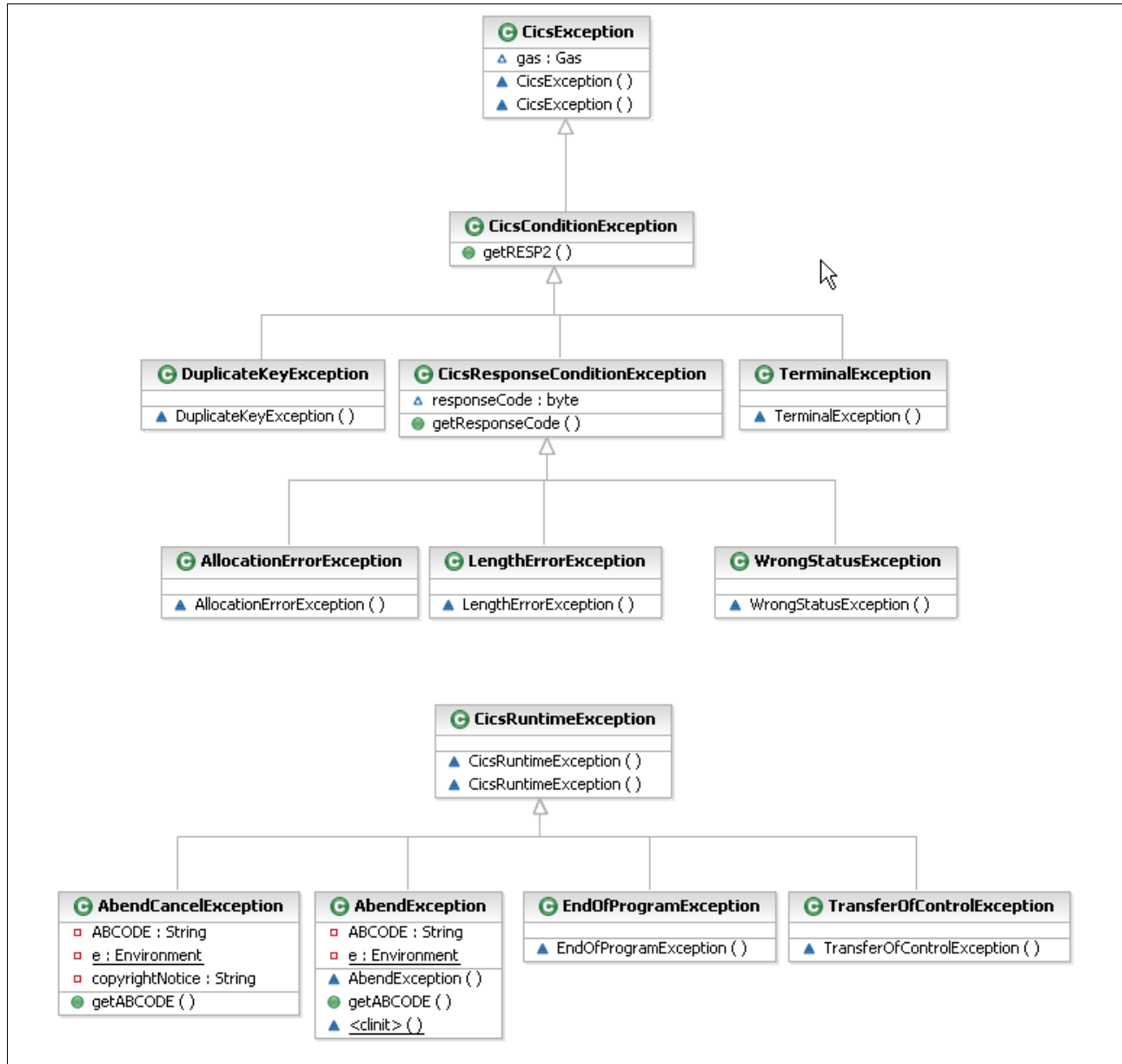


Figure 6-3 Part of the JCICS exception hierarchy

In Java, exceptions fall into two major categories: *Checked exceptions* and *unchecked exceptions*. When you call a method that might throw a checked exception, you are required to either handle the exception in your method or to declare your own method as throwing that exception. Unchecked exceptions, on the other hand, need not necessarily be handled or declared because they usually represent conditions that an application program has difficulty recovering from, and so it is impractical to be forced to handle them.

Example 6-2 on page 98 shows a first example of exception handling in JCICS. The sample program tries to count the number of items in a transient storage queue (we discuss transient storage queues in 6.8, “Using transient storage queues” on page 109). Because there is no straightforward method in the CICS API, we must do this by reading the items in order until CICS responds with an error after trying to read past the last item.

Example 6-2 Exception handling example, first version

```
package com.ibm.itso.sg245275;

import java.io.PrintWriter;

import com.ibm.cics.server.*;

public class ExceptionExample {

    /** Count the number of items in a TS queue. */
    private static int countItems(TSQ tsq) throws InvalidRequestException,
        IOException, LengthErrorException, InvalidSystemIdException,
        ISCInvalidRequestException, NotAuthorisedException,
        InvalidQueueIdException // (1)
    {
        int count = 0;
        ItemHolder item = new ItemHolder();
        try {
            while (true) { // (2)
                tsq.readNextItem(item);
                count++; // (3)
            }
        } catch (ItemErrorException e) { // (4)
            return count;
        }
    }

    public static void main(String[] args) {
        TSQ tsq = new TSQ(); // (5)
        tsq.setName("MYTSQ");
        PrintWriter out = Task.getTask().out;
        try {
            out.println("Number of items in TSQ " + tsq.getName() // (6)
                + ": " + countItems(tsq));
        } catch (InvalidRequestException e) { // (7)
            out.println("Invalid request");
        } catch (IOException e) {
            out.println("I/O error");
        } catch (LengthErrorException e) {
            out.println("Length error");
        } catch (InvalidSystemIdException e) {
            out.println("Invalid system id");
        } catch (ISCInvalidRequestException e) {
            out.println("Inter-system invalid request");
        } catch (NotAuthorisedException e) {
            out.println("Not authorized to access queue");
        } catch (InvalidQueueIdException e) {
            out.println("Invalid queue ID");
        }
    }
}
```

Notes on Example 6-2:

- The `countItems()` method expects a parameter of type `TSQ`, and declares that it might throw several different exceptions—namely, all exceptions that can be thrown by invoking the `TSQ.readNextItem()` method except for the one we handle ourselves.

- ▶ In a *semi-infinite* loop, we try to read the items in the transient storage queue. The loop is not really infinite because it is eventually terminated when no more items are left in the queue.
- ▶ At this point, the call to `TSQ.readNextItem()` succeeded, and we increment the number of items read.
- ▶ When there are no more items left in the queue, JCICS raises an `ItemErrorException`. The loop is terminated and control flow reaches the catch block. We return the number of items read so far.
- ▶ If any other exception occurred, however, we do not catch it in the `countItems()` method; rather, the calling method must handle it, which is why we listed all possible exception classes in the method declaration, except `ItemErrorException`, which is handled in the `countItems()` method itself.
- ▶ The main method creates a TSQ object and sets the queue name.
- ▶ ... and calls `countItems()` to print the number of items in the queue.
- ▶ Because the `countItems()` method declares several checked exceptions, we either must handle them or declare them to be thrown from the `main()` method. In Example 6-2 on page 98, we 'handle' them by listing each possible exception in a catch block and printing a short error message corresponding to the exception type. Obviously, in production code we want to do much more to report the error properly and take action to deal with the problem.

Note that we do not have to catch an `ItemErrorException` because it is already handled in the `countItems()` method. In fact, if we try to handle it, we get a compilation error.

Now, if we have a closer look at the long list of catch clauses in the main method of Example 6-2 on page 98, we see that several exceptions cannot actually be thrown., for example, an `InvalidSystemIdException` or `ISCInvalidRequestException` can only occur when dealing with remote TS queues (that is, they live in a different CICS region). The queue we use in this example, however, is local (we never invoked the `setSysId()` method).

Therefore, in a second version of the example, Example 6-3, we chose to specifically handle only the exceptions that are somewhat likely to occur and handle all others in a generic way, rendering them *unexpected*.

Example 6-3 Exception handling example, second version

```
public static void main(String[] args) {
    TSQ tsq = new TSQ();
    tsq.setName("MYTSQ");
    createItems(tsq);
    PrintWriter out = Task.getTask().out;
    try {
        out.println("Number of items in TSQ " + tsq.getName() + ": " + countItems(tsq));
    } catch (NotAuthorisedException e) {
        out.println("Not authorized to access queue");           // (1)
    } catch (InvalidQueueIdException e) {
        out.println("Invalid queue ID");                         // (2)
    } catch (CicsException e) {
        out.println("Unexpected CICS exception: " + e);          // (3)
    }
}
```

Notes on Example 6-3 on page 99:

- ▶ In this version of the example, we chose to only handle `NotAuthorisedException`.
- ▶ ... and `InvalidQueueIdException`.
- ▶ This catch clause handles all other exceptions. Note that order is important: A compilation error is displayed if a catch clause for a more generic exception textually appears before a catch clause for a more specific one (that is, for one of its subclasses).
All checked exceptions that the JCICS API throws are subclasses of `CicsException`, so this clause handles all of them except the two that we chose to handle specifically.

Exception handling in CICS (or generally in Java, for that matter) is seemingly easy but still often done wrong. It is important that your code does “the right thing,” not only if everything works smoothly, but also in case of failure. In Example 6-4, our intention is to protect a shared resource against concurrent updates using the `NameResource` mechanism (more about `NameResource` in 6.2.3, “Synchronization” on page 92). Basically, a `NameResource` is a global flag that indicates whether some resource is in use.

The code in Example 6-4 looks simple enough: Acquire the lock on the shared resource, perform the update, and release the lock. If an error occurs (either when trying to acquire or release the lock, or when updating), an error message is logged.

Example 6-4 Incorrect exception handling - Resources held

```
// INCORRECT EXCEPTION HANDLING

private void doUpdate() throws CicsException {
    // ... code omitted
}

private void updateSharedData() throws ResourceUnavailableException, LengthErrorException {
    NameResource lock = new NameResource();
    lock.setName("SG245275.LOCK");
    try {
        lock.enqueue();           // Lock the shared resource.
        doUpdate();              // Perform the update.
        lock.dequeue();          // Release the lock.
    } catch (CicsException e) {
        logError("Update failed: " + e);
    }
}
```

However, there is one serious flaw in the code: What happens when the lock is acquired successfully, but the update failed? Control is passed to the catch block, the `lock.dequeue()` call is never executed, and the program still holds the lock. Obviously, that is a bad thing because other instances of the application might want to access the protected resource as well. Of course, this problem is easy enough to fix, you say. Just release the lock in the catch block as well, as shown in Example 6-5.

Example 6-5 Still incorrect exception handling

```
try {
    lock.enqueue();           // Lock the shared resource.
    doUpdate();              // Perform the update.
    lock.dequeue();          // Release the lock.
} catch (CicsException e) {
    logError("Update failed: " + e);
    lock.dequeue();          // Release the lock.
}
```

But that is not much better. First, we have duplicate code, and second, what if the call to `logError()` fails? We still have the same problem. Swapping the two lines in the catch block is not much better either because the `dequeue()` call might fail, and the error is never logged.

The proper way to do it is to use the try-catch-finally mechanism. Code in a *finally* block always gets control, no matter if the corresponding *try* block completed successfully or raised an exception. Example 6-6 shows how this is done.

Example 6-6 Correct exception handling using try/catch/finally

```
private void doUpdate() throws CicsException {
    // ... code omitted
}

private void updateSharedData() throws ResourceUnavailableException, LengthErrorException {
    NameResource lock = new NameResource();
    lock.setName("SG245275.LOCK");
    lock.enqueue();           // Lock the shared resource.
    try {
        doUpdate();           // Perform the update.
    } catch (CicsException e) {
        logError("Update failed: " + e);
    } finally {
        lock.dequeue();       // Release the lock.
    }
}
```

In Example 6-6, the call to `lock.dequeue()` is in a finally block, and is therefore executed regardless if the `doUpdate()` call or the `logError()` call succeeded or failed, which ensures that the lock is released in any event. Also, we no longer have duplicate code.

6.6 Calling other programs and passing data

In traditional CICS programming, it is common to split an application into several programs. Each program handles a specific part of the application, for example, you can have one front-end program that handles user interaction and one or more subprograms that handle the business logic, such as retrieving data from and inserting data into a database or VSAM file. The front-end program displays menus and data entry forms (*maps* in CICS parlance), possibly also for input verification, and calls one of the business logic subprograms, depending on the user's input.

To call another program, you use the CICS LINK mechanism. A LINK causes the invoking program to be temporarily suspended and the linked-to program to be invoked. When the linked-to program returns, control is given back to the original program. The invoked program is still part of the same unit of work. Therefore, any work that is done to recoverable resources by both programs is backed out if the task does not complete successfully. In other words, a CICS LINK works just like a subroutine call except that the calling and called programs are not statically linked together.

To pass control to another program, use the XCTL (transfer control) command. The program issuing an XCTL does not receive control back when the target program terminates; rather, the program that is one level up in the call hierarchy does. See Figure 6-4 on page 102 for an illustration of the program control flow with LINK and XCTL.

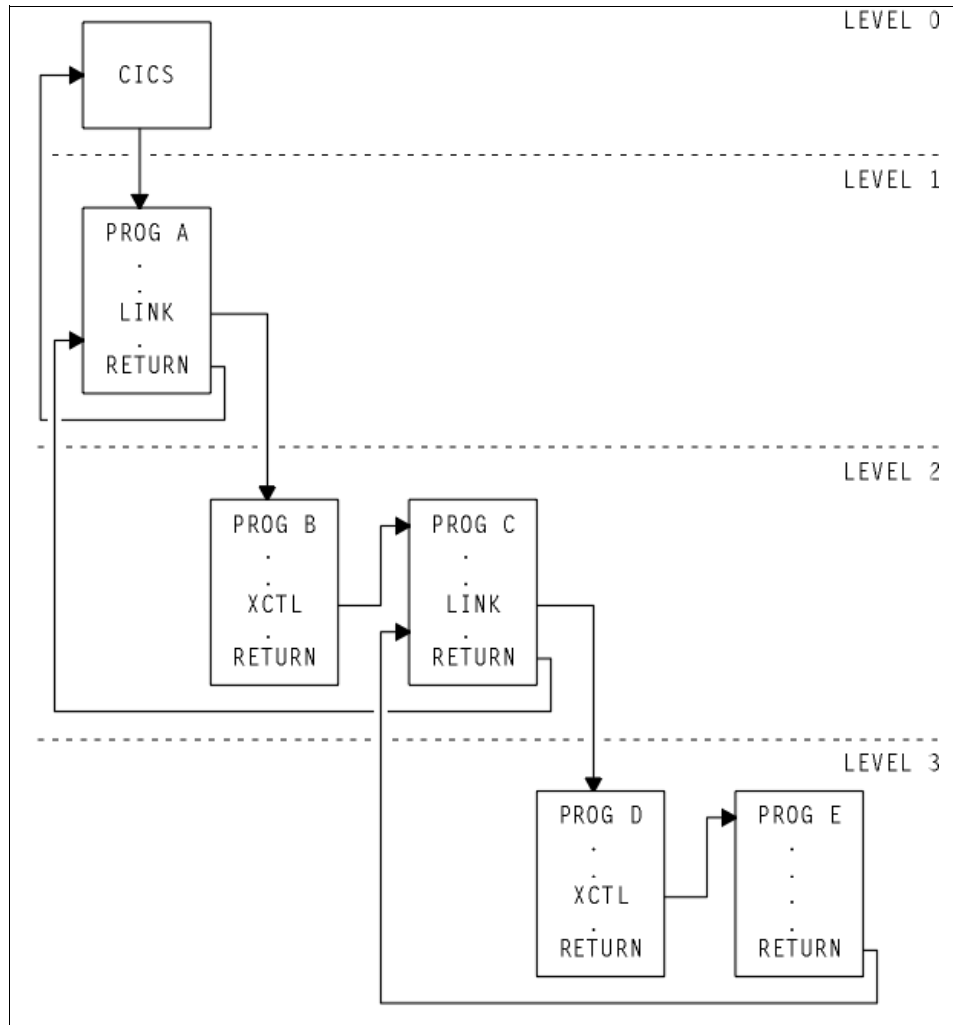


Figure 6-4 Program control flow with CICS LINK and XCTL

Of course, in most cases, the linking program must communicate with the linked-to program, for example, to pass in a function code that tells it what specific function to perform and to get results back from it. In CICS, this is done either using the **COMMAREA** mechanism or using the newer **Container** mechanism. A **COMMAREA** is simply a storage area that is owned by the linking program and is made available to the linked-to program. Containers are similar, but they offer advantages over **COMMAREAs** in certain circumstances, particularly if larger quantities of data must be passed.

In CICS Java applications, separation of logic into several distinct subprograms is not that common, primarily because the Java language lends itself well enough to separate different aspects of the application into different parts of the code, namely, into different classes. Calling separate programs using **LINK** or **XCTL** is also more expensive than internal calls between methods in Java programs.

In practice, however, you probably must call existing programs from your Java code, for example, if you want to access heritage modules that are too expensive to be reimplemented using Java.

6.6.1 Calling other programs using LINK and XCTL

The Java equivalents to EXEC CICS LINK and EXEC CICS XCTL are the methods `Program.link()` and `Program.xctl()`, respectively. There are several versions of each method, taking different parameters, depending on whether and how you want to pass a COMMAREA to the called program. So, to set up a LINK (or XCTL) to another program, you create an instance of class *Program*, call its `setName()` method to supply the name of the program you want to transfer control to, and call `link()` or `xctl()` as appropriate.

The usual pattern is to create and initialize the *Program* object in a constructor so that it can be used from your individual methods, as shown in Example 6-7.

Example 6-7 Typical usage of Program.link()

```
public class LinkDemo {

    private final Program okk850;

    public LinkDemo() {
        okk850 = new Program();
        okk850.setName("OKK850");
    }

    public void callOKK850() throws InvalidRequestException, LengthErrorException,
        InvalidSystemIdException, NotAuthorisedException, InvalidProgramIdException,
        RolledBackException, TerminalException
    {
        okk850.link();
    }

    ...

}
```

6.6.2 Passing data between programs

To pass data between a calling program and a called program, CICS programs typically use the COMMAREA mechanism. Since CICS Version 3.1, they have the alternative of using the Container mechanism. These two mechanisms have much in common, so we provide an overview of both first and then a more detailed discussion of each follows.

A COMMAREA is nothing more than an area of storage that is owned by the calling program and is made available to the callee. A COMMAREA is limited to a maximum of 32 kB of storage, while Containers do not have this restriction.

Containers are named blocks of data that are designed to pass information between programs. You can think of them as “named COMMAREAs”. Programs can pass any number of containers between each other and can pass a theoretically unlimited amount of data. Containers are grouped together in sets called *channels*. A channel is analogous to a parameter list.

CICS does not care about the format or layout of the COMMAREA or Container, which is entirely up to the two programs that are involved. Typically, the format is defined in a host language data structure, such as a COBOL Level-01 record or a C struct.

In JCICS, there are six different varieties of the `Program.link()` method:

- ▶ `Program.link()`
Performs a link without a COMMAREA.
- ▶ `Program.link(byte[] commarea)`
Performs a link with a COMMAREA. The linked-to program can modify the COMMAREA.
- ▶ `Program.link(byte[] commarea, int datalength)`
Performs a link with a COMMAREA, where only the first `datalength` bytes are passed to the linked-to program. The linked-to program can modify the entire COMMAREA, even if it is longer than `datalength` bytes.
- ▶ `Program.link(Channel chan)`
Performs a link to the program that is passing a CHANNEL.
- ▶ `Program.link(ByteBuffer commarea)`
`Program.link(ByteBuffer in, ByteBuffer out)`

These methods were retained for compatibility only. They were supplied for integration with the IBM Record Framework library, which came with VisualAge® for Java. Because the Record Framework was not carried over to Rational Application Developer, they can be considered deprecated.

Tip: If you encapsulate the mapping of data fields to a byte array into a single Java class, with getters and setters for all of the fields, you can use this class in both the sending and receiving programs to reduce the risk of the two programs getting out of sync with each other. This is similar to using a shared copybook in communicating COBOL programs.

6.6.3 Communicating using the COMMAREA

We demonstrate the COMMAREA mechanism with a simple program that just reverts the COMMAREA being passed, as shown in Example 6-8.

Example 6-8 Program to revert the passed COMMAREA

```
package com.ibm.itso.sg245275;

import com.ibm.cics.server.CommAreaHolder;

public class Revert {

    public static void main(CommAreaHolder cah) {
        byte[] data = cah.value;
        int n = data.length - 1;
        for (int i = (n - 1) / 2; i >= 0; --i) {
            byte temp = data[i];
            data[i] = data[n - i];
            data[n - i] = temp;
        }
    }
}
```

To test the program, we write another little program, Example 6-9 on page 105, which invokes it using `Program.link()`.

Example 6-9 Linking to the Revert program

```
public class RevertTest {
    private static String revert(String s) throws CicsException {
        Program revert = new Program();
        revert.setName("REVERT");
        byte[] bytes = s.getBytes();
        revert.link(bytes);
        return new String(bytes);
    }

    public static void main(CommAreaHolder cah) {
        PrintWriter out = Task.getTask().out;
        try {
            String original = "Hello World!";
            String reverted = revert(original);
            out.println();
            out.println("Original: " + original);
            out.println("Reverted: " + reverted);
        } catch (CicsException ex) {
            out.println("Oops: " + ex);
        }
    }
}
```

Alternatively, you can use the CECI transaction to test it, as shown in Example 6-10.

Example 6-10 Testing the Revert program using CECI

```
link program(REVERT) commarea('The quick brown fox')
STATUS:  COMMAND EXECUTION COMPLETE                                NAME=
EXEC CICS LINK Program( 'REVERT ' )
  < Commarea( 'xof nworb kciuq ehT' ) < Length( +00019 ) > < Datalength() > >
  < SYSid() >
  < SYNconreturn >
  < Transid() >
  < INPUTMSG() < INPUTMSGLen() > >

RESPONSE: NORMAL                                EIBRESP=+0000000000 EIBRESP2=+0000000000
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

6.6.4 Communicating through Channels and Containers

We demonstrate the use of Channels and containers by changing the COMMAREA example program to use channels and containers instead of a COMMAREA, as shown in Example 6-11.

Example 6-11 Program to revert the data passed in a container

```
public class Revert2 {

    public static void main(CommAreaHolder cah) {
        Task task = Task.getTask();
        Channel channel = task.getCurrentChannel();                // (1)
        if (channel != null) {
            try {
                Container record = channel.getContainer("Revert_Record");    // (2)
            }
        }
    }
}
```

```

        byte[] data = record.get();                                // (3)

        int n = data.length - 1;
        for (int i = (n - 1) / 2; i >= 0; --i) {
            byte temp = data[i];
            data[i] = data[n - i];
            data[n - i] = temp;
        }
        record.put(data);                                          // (4)
    }
    catch (ContainerErrorException e) {                            // (5)
        System.out.println("Container Error Exception "+
            "- probably container not found");
        e.printStackTrace();
    }
    catch (ChannelErrorException e) {
        System.out.println("Channel Error Exception "+
            "- probably invalid channel name or write to a read-only channel");
        e.printStackTrace();
    }
    catch (InvalidRequestException e) {
        System.out.println("CICS INVREQ condition "+
            " - Something has gone badly wrong!");
        e.printStackTrace();
    }
    catch (CCSIDErrorException e) {
        System.out.println("CICS CCSIDERR condition "+
            " - Code page conversion problem");
        e.printStackTrace();
    }
}
else {
    System.out.println("There is no Current Channel");
}
}
}

```

There are several differences between this program and the program that uses a COMMAREA:

1. We first must get the channel from the task, which allows us to get to the data that is passed to this program.
2. After we have the channel, we can get the container that has the data that was passed to the program. In a more complex program, we can retrieve multiple containers from the channel and process each accordingly.
3. As with a COMMAREA, we can get the data as a byte array and process it in the same manner.
4. For simplicity, we re-used the container to return the data to the calling program, but it is advisable to use a new container to return the data.
5. There is a selection of exceptions that can be thrown when you work with containers. These exceptions map closely to the CICS return conditions that can occur when you use channels and containers in other languages. This program attempts to provide some indication of what the problem was that caused the exception, but it does not attempt to recover from the problem. In a more sophisticated program, you want to do more to recover from these exceptions. It is also possible to catch most or all of the exceptions with a more generic catch block, such as catch 'CICSEException', but this reduces the opportunity to deal with different types of exceptions appropriately.

Once again, we can test the program with a small program, shown in Example 6-12, which is similar to the previous one and also uses `Program.link()` but this time passing a container.

Example 6-12

```
public class Revert2Test {
    private static String revert(String s) throws CicsException {
        Program revert2 = new Program();
        revert2.setName("REVERT2");
        Task task = Task.getTask();
        Channel data = task.createChannel("Revert_Data");           // (1)
        Container record = data.createContainer("Revert_Record");    // (2)
        record.put(s);                                              // (3)
        revert2.link(data);                                         // (4)
        byte[] bytes = record.get();
        return new String(bytes);
    }

    public static void main(CommAreaHolder cah) {
        PrintWriter out = Task.getTask().out;
        try {
            String original = "Hello World!";
            String reverted = revert(original);
            out.println();
            out.println("Original: " + original);
            out.println("Reverted: " + reverted);
        } catch (CicsException ex) {
            out.println("Oops: " + ex);
        }
    }
}
```

Again, there are differences between this program and the one using a COMMAREA:

1. We create a channel, and give it a name.
2. After we have a channel, we can create a container to hold the data that we want to pass to the program that is being called. A more complex program might create multiple containers on a single channel to be passed to a target program.
3. The data can be placed in the container either as a byte array or as a String. If a String is passed to the container, it is automatically converted to a byte array using the default encoding in the JVM.
4. The call to the other program is passed the reference to the channel that owns the container.

You can run this program in exactly the same way that you ran the previous test program, and the output is exactly the same.

6.6.5 COMMAREAs versus channels and containers

Channels and containers offer many benefits over COMMAREAs, especially where large (greater than 32kB) quantities of data must be passed. A full discussion of these advantages, and how best to make use of them would fill an entire book, and fortunately, that book already exists in the form of the existing IBM Redbooks publication, *CICS Transaction Server V3R1 Channels and Containers Revealed*, SG24-7227-00.

As a simple rule-of-thumb, the following rules can help:

- ▶ Where the application must pass large quantities of data, channels and containers are easily the best choice.
- ▶ For existing applications that use COMMAREAs, and pass small quantities of data, the effort of converting the program might not be justified.
- ▶ For new applications, channels and containers are typically the default choice.

Obviously, for the full benefit of channels and containers to be realized, effort must be applied in their correct usage in the application and in appropriate system configuration. *CICS Transaction Server V3R1 Channels and Containers Revealed*, SG24-7227-00 offers considerable guidance in both these areas.

6.7 Remoteable resources

Most CICS resources, such as files, TS queues, and programs, need not actually reside in the CICS region from which they are accessed, but can live in another region connected to it. Resources can either be set up in a CICS region to be remote, in which case the program using the resource does not even know that the resource is owned by a different region, or a program can explicitly access a resource in another region.

In JCICS, remoteable resources are subclasses of the abstract class `RemoteableResource` (see Figure 6-5 on page 109). This class has two methods, `setSysId()` and `getSysId()`, to respectively set and retrieve the name of the region that owns the resource. All of the JCICS resource classes, except `WebService`, extend `RemoteableResource` and so can be considered to be Remoteable.

The region accessing the resource and the region owning the resource must be set up to “know each other” in the network. We do not explain the details of setting up a connection between CICS regions in this publication; however, *CICS Intercommunication Guide*, SC34-6243 has the details, or ask your CICS system programmer.

Example 6-13 demonstrates how to start a new transaction in a different CICS region, using the `StartRequest` class, which we briefly discussed in 6.12, “Interval control” on page 124.

Example 6-13 Starting a transaction in another CICS region

```
StartRequest startRequest = new StartRequest();
startRequest.setName("TRN2");           // Name of transaction to be started
try {
    startRequest.setSysId("PJA6");       // System ID of remote CICS region
    startRequest.issue();                // Issue request
} catch (CicsException e) {
    log("Error starting remote transaction: " + e);
}
```

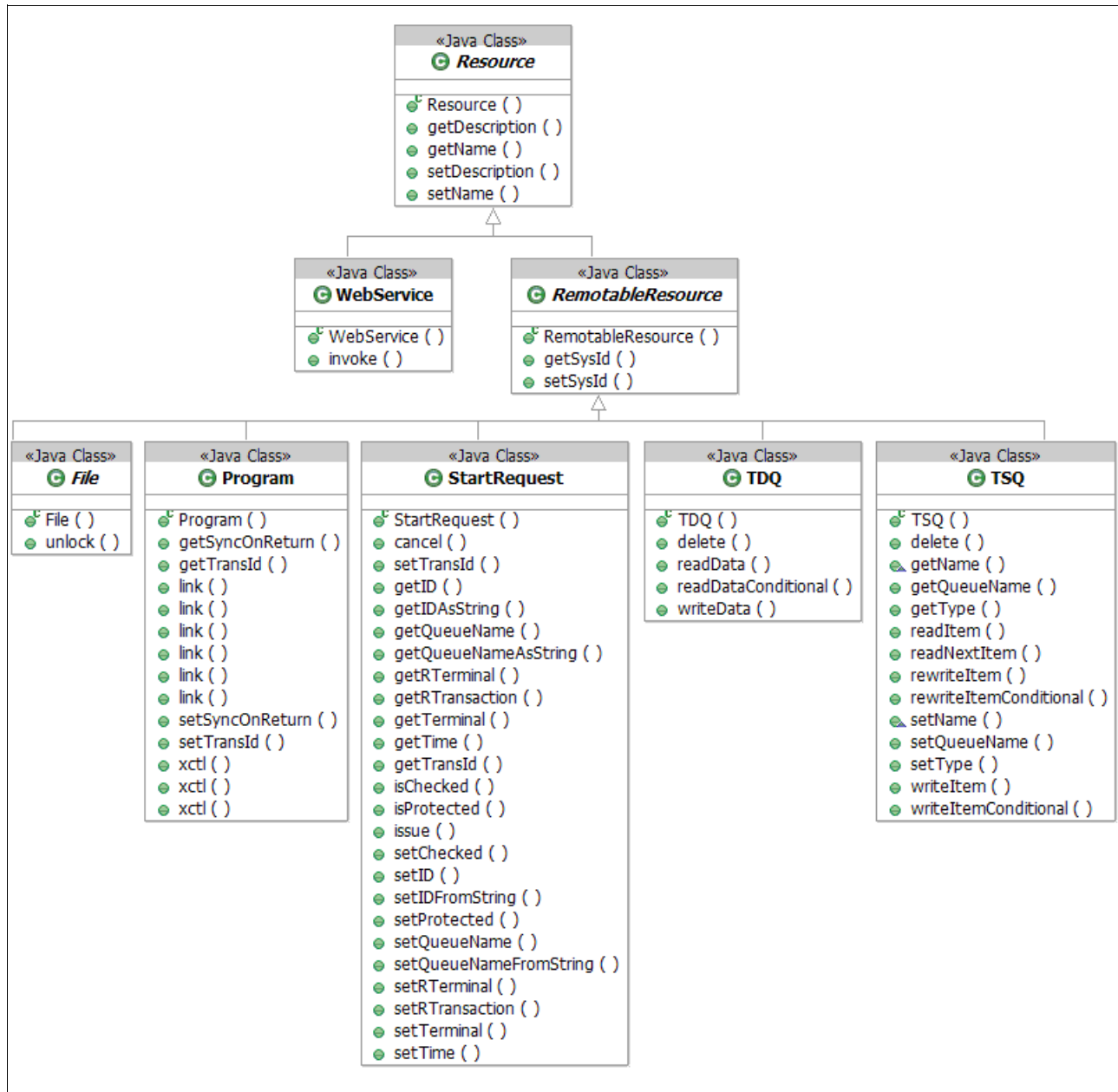


Figure 6-5 The Resource class hierarchy

6.8 Using transient storage queues

Transient storage queues (TS queues for short) are a simple mechanism for interprocess communication in CICS. Basically, a TS queue is simply an array of numbered slots or items. You can either add a new item to a queue or replace an existing item that is identified by its slot number. The maximum number of items is 32767, and each item can hold up to 32763 bytes of data.

One additional benefit of TS queues, other than being easy to use, is that they can be created dynamically. In other words, no additional set up is necessary in CICS to create a TS queue—if it did not exist before, it is automatically created when a new item is added to it.

A TS queue might also be predefined if the queue is to have special characteristics, such as security, recoverability, or is remote.

A common use of a TS queue is a *scratchpad* functionality, which is a shared storage area that is available to all instances of an application running in the CICS region. In this section, we develop a simple Scratchpad class to demonstrate usage of TS queues with JCICS.

Example 6-14 shows a skeleton version of our Scratchpad class, just having a reference to the TS queue to use, and three constructors.

Example 6-14 Skeleton Scratchpad class

```
package com.ibm.itso.sg245275;

// Import declarations omitted...

/**
 *
 * Demonstrates how to use TS queues as a scratchpad.
 *
 * @author Ulrich Gehlert
 */
public class Scratchpad {

    /** Default TS queue name. */
    public static final String DEFAULT_SCRATCHPAD_TSQ_NAME = "SCRATCH";    // (1)

    /** The TS queue we are writing to. */
    private final TSQ tsq;    // (2)

    /** Constructor using the default TS queue name. */
    public Scratchpad() {    // (3)
        this(DEFAULT_SCRATCHPAD_TSQ_NAME);
    }

    /** Constructor using an explicit TS queue name. */
    public Scratchpad(String tsqName) {    // (4)
        this(new TSQ());
        tsq.setName(tsqName);
    }

    /** Constructor using an explicit TS queue (which may be remote). */
    public Scratchpad(TSQ tsq) {    // (5)
        this.tsq = tsq;
    }
}
```

Notes on Example 6-14:

- ▶ This is the default queue name if the user of the class chose to use the no-argument constructor.
- ▶ The TS queue where the data goes to. The variable is declared `final`, so it must be initialized, either directly or indirectly, by each constructor.
- ▶ The no-argument constructor sets up a TS queue with the default name.
- ▶ Constructor taking an explicit TS queue name.
- ▶ Constructor taking an initialized TS queue object. Callers use this constructor if they want to write to a remote TS queue, for example, because they want the scratchpad to be available across CICS regions.

Next, we implement methods for writing data into and getting data from the scratchpad, as shown in Example 6-15 and Example 6-16, respectively.

Example 6-15 Writing data to the scratchpad

```
/**
 * Write a byte array into the scratchpad queue.
 *
 * @param bytes The byte array to be written.
 */
public void writeBytes(byte[] bytes) throws ItemErrorException, InvalidRequestException,
    IOException, LengthErrorException,
    InvalidSystemIdException, ISCInvalidRequestException, NotAuthorisedException,
    InvalidQueueIdException
{
    try {
        tsq.rewriteItem(1, bytes); // (1)
    } catch (InvalidQueueIdException e) {
        // The queue didn't exist -- add item to queue, thus creating the queue
        tsq.writeItem(bytes); // (2)
    }
}

/**
 * Write a string to the scratchpad.
 *
 * @param s The string to be written to the scratchpad.
 */
public void writeString(String s) throws InvalidRequestException, IOException,
    LengthErrorException,
    InvalidSystemIdException, ItemErrorException, ISCInvalidRequestException,
    NotAuthorisedException, InvalidQueueIdException
{
    writeBytes(s.getBytes()); // (3)
}
```

Notes on Example 6-15:

- ▶ In Example 6-15, we never write into any TS queue slots other than the first one. Note that the slot numbers count from 1, not from 0.
- ▶ If we got an `InvalidQueueIdException` when trying to write to the TS queue, it did not exist. So, we use the `writeItem()` method, which causes the queue to be created.
- ▶ There is a possible race condition: Another task can create the queue right between our failed call to `rewriteItem()` and our call to `writeItem()`. We can solve the problem by protecting this section of code with a `NameResource`. For the sake of brevity, we ignored that possibility.
- ▶ This convenience method allows us to write strings into the queue by converting a *String* to a `byte[]` and writing that byte array to the queue. We implement a corresponding `readString()` method to read a *String* back. Of course, users of the *Scratchpad* class must agree upon what kind of data to put on the scratchpad.

Example 6-16 shows an example of reading data from the scratchpad.

Example 6-16 Reading data from the scratchpad

```
/**
 * Read a byte array from the scratchpad.
 *
 * @return The byte array on the scratchpad, or <code>null</code> if the
```

```

        *          scratchpad is empty.
        */
public byte[] readBytes() throws InvalidRequestException, IOException,
    LengthErrorException, InvalidSystemIdException,
    ISCInvalidRequestException, NotAuthorisedException
Writing data {
    try {
        ItemHolder item = new ItemHolder();
        tsq.readItem(1, item);
        return item.value;
    } catch (InvalidQueueIdException e) {
        return null;
    } catch (ItemErrorException e) {
        return null;
    }
}

/**
 * Read a string from the scratchpad.
 *
 * @return The string currently on the scratchpad.
 */
public String readString() throws InvalidRequestException, IOException,
    LengthErrorException, InvalidSystemIdException,
    ISCInvalidRequestException, NotAuthorisedException, InvalidQueueIdException
{
    byte[] bytes = readBytes();
    if (bytes == null)
        return null;
    else
        return new String(bytes);
}

```

6.9 Performing serialization

When you want to make sure that a given resource cannot be accessed by more than one task at a time because you need some form of serialization mechanism.

A resource, in this context, can be a physical resource, such as a TS queue or a file, or it might be a *virtual* resource, such as a credit card number.

A well-known solution to ensure mutual exclusion is the semaphore mechanism, which in z/OS and CICS is referred to as ENQ/DEQ. A semaphore is simply an integer variable (a counter) with an associated queue. When a task issues an ENQ for a given resource, other tasks that try to ENQ on the resource are suspended until the first task releases the resource using a DEQ operation.

A resource in the context of this command is any string of one through 255 bytes, established by in-house standards, to protect against conflicting actions between tasks or to cause single threading within a program. Therefore if you enqueue on the name of a file, another task attempting the enqueue() call on the same file name is unsuccessful, which does not stop some other task from using the file. It means that the enqueue request is unsuccessful if another task attempts an enqueue on the same character string, which in this case happens to be the same as the string of characters that make up the name of the file. If more than one enqueue() call is issued for the same resource by a given task, the resource remains owned

by that task until the task issues a matching number of dequeue() calls, finishes its current unit of work, or terminates.

You can enqueue by string or by address. The JCICS classes provide an AddressResource class and a NameResource class, as shown in see Figure 6-6.



Figure 6-6 JCICS synchronization support classes

The AddressResource class is supplied for compatibility with other languages that use an enqueue on address (or, in general, on any resource name that is binary). The address that is used must be passed to your Java program through a COMMAREA, TSQ, or other shared area. What you actually supply to the AddressResource class is a byte array.

The preferable enqueue method is the enqueue by (human-readable) name, that is, by string. From your Java program, you create a NameResource object and use the setName() method to supply a 1–255 character string. Then issue the enqueue() method.

An enqueue is held only until the current unit of work finishes, that is, it is automatically released when you call Task.getTask().commit() or Task.getTask().rollback(). Also, it is released if a task ends abnormally.

Restriction: The EXEC CICS ENQ command has an option that allows a task to hold an enqueue across unit-of-work boundaries (ENQ RESOURCE(...) TASK). Unfortunately, there is no corresponding option, or parameter, in the JCICS API.

In z/OS, each ENQ has a given scope that determines the *visibility* of the ENQ. An ENQ might be visible:

- ▶ At local scope, that is, only in the address space issuing it (for CICS applications, the CICS region).
- ▶ At system scope, that is, only in the z/OS image where the program is executing.
- ▶ At sysplex scope, that is, in the entire sysplex.

In CICS, an application program cannot by itself determine the scope of an ENQ. By default, each ENQ has local scope only. If you want to ensure serialization across multiple CICS regions, you must set up a special CICS resource called an ENQMODEL. For details, see *CICS System Definition Guide*, SC34-6226.

Set up an ENQMODEL definition: This is so important that we say it again. An ENQ is local to the CICS region by default. To achieve serialization across multiple regions, possibly running on different z/OS images in the sysplex, you must set up an ENQMODEL definition.

6.10 Web, TCP/IP, and document services

CICS Web support is a collection of CICS services that support direct access to CICS application programs from Web browsers. You can use CICS Web support with:

- ▶ Web-aware application programs that use the EXEC CICS WEB and EXEC CICS DOCUMENT application programming interfaces. See the CICS Application Programming Guide for more information.
- ▶ Programs that are designed to communicate with 3270 terminals using BMS.
- ▶ Programs that are designed to be linked to from another program using a COMMAREA interface.

Applications can use the facilities of the CICS Web support to:

- ▶ Map a URL to a request for CICS services.
- ▶ Interpret HTTP requests.
- ▶ Construct HTTP responses.
- ▶ Build HTML output for display by a Web browser.

Although CICS Web support is designed primarily to provide communication between a Web browser and CICS using HTTP, it also supports clients that send non-HTTP requests. The same components of CICS Web support are used to process HTTP and non-HTTP requests.

CICS has a facility that allows you to build up formatted data areas that are known as documents. Some examples of how these formatted areas, or documents, can be used, are:

- ▶ Constructing a COMMAREA
- ▶ Creating standard formats for printing (for example, using your own letterhead, address, and so on)
- ▶ Sending HTML data to be displayed by a Web browser

We use CICS document support in Chapter 7, “Evolving a heritage application using Java” on page 135, for the latter purpose, namely, to populate predefined HTML templates.

6.11 File control

Using the JCICS file control methods you can query and manipulate *key-sequenced data set* (KSDS), *entry-sequenced data set* (ESDS), and *relative record data set* (RRDS) VSAM files:

- ▶ Key-sequenced data set (KSDS)

A key-sequenced data set has each of its records identified by a key. The key of each record is simply a field in a predefined position within the record. Each key must be unique in the data set.

- ▶ Entry-sequenced data set (ESDS)

An entry-sequenced data set is one in which each record is identified by its relative byte address (RBA), that is, by the byte offset of its starting position within the file. Records are held in an ESDS in the order in which they were first loaded into the data set.

New records added to an ESDS always go after the last record in the data set. You might not delete records or alter their lengths. After a record is stored in an ESDS, its RBA remains constant. When browsing, records are retrieved in the order in which they were added to the data set.

- ▶ Relative record data set (RRDS)

A relative record data set has records that are identified by their relative record number (RRN). The first record in the data set is RRN 1, the second is RRN 2, and so on. Records in an RRDS can be fixed or variable length records, and the way in which VSAM handles the data depends on whether the data set is a fixed or variable RRDS. A fixed RRDS has fixed-length slots predefined to VSAM into which records are stored. The length of a record on a fixed RRDS is always equal to the size of the slot. VSAM locates records in a fixed RRDS by multiplying the slot size by the RRN, which you supply on the file control request, to calculate the byte offset from the start of the data set.

A variable RRDS, on the other hand, can accept records of any length up to the maximum for the data set. In a variable, RRDS VSAM locates the records by means of an index.

A fixed RRDS generally offers better performance. A variable RRDS offers greater function.

The operations that are available on files fall into the following categories, all of which are supported by the JCICS API:

- ▶ Adding new records
- ▶ Reading an existing record
- ▶ Deleting an existing record
- ▶ Changing an existing record
- ▶ Browsing the file

Figure 6-7 on page 116 shows the JCICS class hierarchy for file control classes.

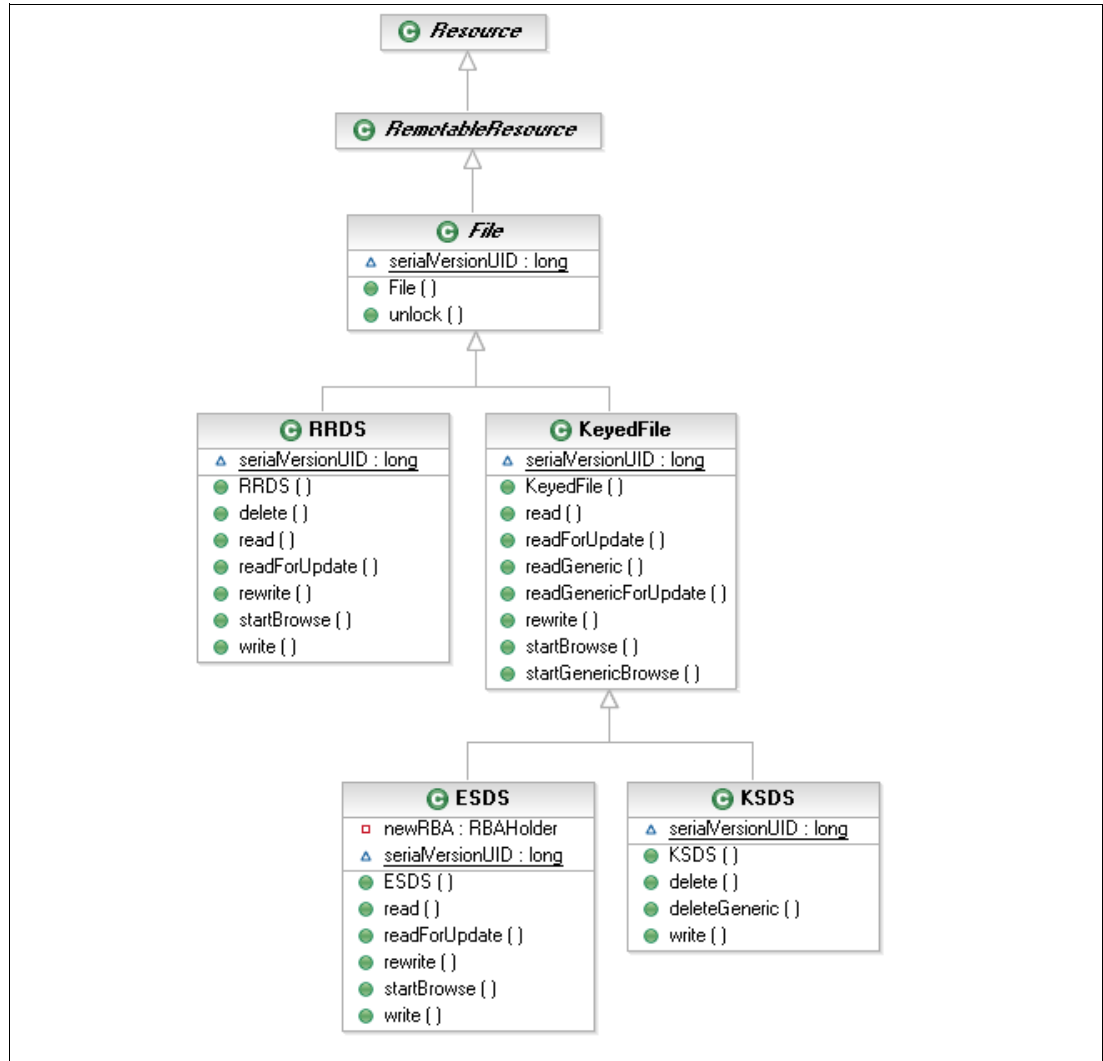


Figure 6-7 JCICS classes for file control

Here we only discuss KSDS file access. Using the other two types of file with JCICS is fairly similar.

In the rest of this section, we develop an implementation of the Java *Map* interface that is backed by a KSDS data set. Essentially, it is a kind of *wrapper* around the KSDS that allows manipulation of the data set in a way that looks more natural to Java programmers than raw JCICS API calls. This example is rather long, and many of the implementation details have nothing to do with JCICS per se, but we feel that it illustrates file access in JCICS quite nicely and can prove to be a useful starting point for your own code.

If you are familiar with Java collection classes, you know that a map is a collection of name/value pairs. You can associate a value with a key (put), delete an association (remove), retrieve the value associated with a key (get), and iterate over the map in three different ways (keys only, values only, or key/value pairs).

The implementation does not always fulfill the Map contract, for example, the documentation for Map says that the remove() method returns the value previously associated with the key or null if no value was previously associated with the key.

However, we chose not to return the previous value but just a Boolean that indicates there actually was a record in the file with the given key (or null if there was none) because of performance: If we want to return the previously associated value, we must perform an extra read operation. Because the value returned from `remove()` is not typically used anyway, we chose to consciously break the contract.

Class declaration and constructors

We start with the class declaration, which includes fields and constructors, as shown in Example 6-17.

Example 6-17 VsamMap class skeleton

```
package com.ibm.itso.sg245275.vsam;

import java.util.AbstractMap;
... more import statements ...

/**
 * An implementation of a Map that is backed by a
 * KSDS VSAM file.
 *
 * Note that, for simplicity and performance, this
 * implementation violates some of the Map interface's contracts.
 * For example, the {@link #remove(Object) remove()} method
 * does not return the value previously associated with the key.
 *
 * @author Ulrich Gehlert
 */
public class VsamMap extends AbstractMap {                                // (1)
    private class VsamIterator implements Iterator {                    // (2)
        ... see below ...
    }
    static class SimpleEntry implements Entry {
        ... not shown here (copied from java.util.AbstractMap)
    }
    /** Length of key. */
    private final int keylength;

    /** Underlying KSDS. */
    private final KSDS ksds;                                            // (3)

    private final RecordHolder recordHolder = new RecordHolder();

    /**
     * Constructor.
     */
    public VsamMap(KSDS ksds, int keylength) {                          // (4)
        this.ksds = ksds;
        this.keylength = keylength;
    }

    /**
     * Convenience constructor that accepts a KSDS name.
     */
    public VsamMap(String ksdsName, int keylength) {                    // (5)
        this(new KSDS(), keylength);
        ksds.setName(ksdsName);
    }
}
```

Notes on Example 6-17 on page 117:

- ▶ Rather than implement all methods of the Map interface ourselves, we extend the AbstractMap class, which already provides a lot of the implementation.
- ▶ In fact, the only abstract method in AbstractMap is entrySet(), so in theory we only need to implement that. However, the default implementations of several other operations are not terribly efficient, so we override them (for example, AbstractMap.get() iterates over all associations until it finds an entry with the given key).
- ▶ Eventually, this is an implementation of Iterator, which iterates over the entries in the KSDS (see Example 6-23 on page 121).
- ▶ This is the object that represents the VSAM KSDS by which this map is backed.
- ▶ This constructor takes a KSDS and a key length. We need the key length later for iterating through the file.
- ▶ For convenience, this is a constructor that takes a String argument (the KSDS name) rather than a pre-initialized KSDS object.

Getting a record

Next, we implement the get() method, Example 6-18, which accepts a key and returns the associated value or null when no value is associated with the key (that is, if there is no record in the file having that key). Also, it is easy to implement containsKey() using get().

To get the value that is associated with the key, we try to read a record with the given key from the underlying VSAM KSDS.

Example 6-18 VsamMap.get()

```
public Object get(Object key) {
    try {
        ksds.read(keyToBytes(key), recordHolder);           // (1)
        return valueFromBytes(recordHolder.value);          // (2)
    } catch (RecordNotFoundException notfound) {
        return null;                                         // (3)
    } catch (CicsException e) {
        throw new RuntimeException(e);                       // (4)
    }
}

public boolean containsKey(Object key) {
    return get(key) != null;                                // (5)
}
```

Notes on Example 6-18:

- ▶ Read a record with the given key from the KSDS. The first argument to KSDS.read() is the key, as a byte array. The second argument is a RecordHolder object, which contains the data after a successful read.
- ▶ Because, in general, our keys and records do not have the form of byte arrays, we call helper methods to convert keys and records to byte arrays, and vice versa. The default implementations (see Example 6-19 on page 119) assume that both keys and records are String objects. Subclasses can override these methods to provide their own conversions.
- ▶ The KSDS read was successful. We return the record after converting it.
- ▶ When the KSDS did not contain a record with the given key, JCICS raises a RecordNotFoundException. Because the Map interface states that get() return null when no value is associated with the key, we catch the exception and do so.

- All other exceptions are wrapped into a `RuntimeException`. We cannot directly throw instances of `CicsException` because these are checked exceptions.
- Implementing `containsKey()` is straightforward. Check if `get()` returns non-null.

The next piece of code, in Example 6-19 provides default implementations of the conversion methods that we talked about in the notes on Example 6-18 on page 118.

Example 6-19 VsamMap - Conversion methods

```
protected byte[] keyToBytes(Object key) {
    return ((String) key).getBytes();
} // (1)

protected byte[] valueToBytes(Object value) {
    return ((String) value).getBytes();
}

protected Object keyFromBytes(byte[] bytes) {
    return new String(bytes);
} // (2)

protected Object valueFromBytes(byte[] bytes) {
    return new String(bytes);
}
```

Notes on Example 6-19:

- This default implementation assumes that both keys and records are strings. So, to convert the key to a byte array, we cast it to a `String` and convert that string to a byte array.
- To convert back, we construct a new `String` object from the byte array.

As mentioned before, these methods are default implementations only and are meant to be overridden by subclasses as appropriate (that is why they were declared *protected*).

Removing a record

Next, we implement `remove()`, which deletes a record from the underlying VSAM data set, as shown in Example 6-20.

Example 6-20 VsamMap.remove()

```
public Object remove(Object key) {
    try {
        ksds.delete(convertToBytes(key));
        return Boolean.TRUE;
    } catch (RecordNotFoundException e) {
        return null;
    } catch (CicsException e) {
        throw new RuntimeException(e);
    }
}
```

Notes on Example 6-20 on page 119:

- Again, we convert the key to a byte array, and then we call `KSDS.delete()`.

- ▶ We return Boolean.TRUE if the record was actually deleted, which is a deliberate violation of the Map contract, which says that delete() returns the value previously associated with the key. Although fairly easy to implement, this causes another CICS call, which degrades performance.
- ▶ If the record was not found, return null.

Adding a record

To implement Map.put(), we try to write a record to the data set, as shown in Example 6-21.

Example 6-21 VsamMap.put()

```
public Object put(Object key, Object value) {
    byte[] keyBytes = convertToBytes(key);
    byte[] valueBytes = convertToBytes(value);
    try {
        ksds.write(keyBytes, valueBytes);                // (1)
    } catch (DuplicateRecordException e) {
        throw new IllegalArgumentException("Duplicate key"); // (2)
    } catch (CicsException e) {
        throw new RuntimeException(e);
    }
    return null;
}
```

Notes on Example 6-21:

- ▶ Try to write the record to the KSDS.
- ▶ If we receive a DuplicateRecordException, we throw an IllegalArgumentException, which is explicitly permitted by the documentation of java.util.Map (“some aspect of this key or value prevents it from being stored in this map”).
- ▶ So, effectively, this implementation only allows you to add new records but not to modify existing ones.

Iteration

Finally, we implement Map.entrySet(), which is the only abstract method in AbstractMap, as shown in Example 6-22, which adds support for iterating over the keys or values in the Map, in other words, to browse the file.

Example 6-22 VsamMap.entrySet()

```
public Set entrySet() {
    return new AbstractSet() {
        public Iterator iterator() {
            return new VsamIterator();
        }

        public int size() {
            throw new UnsupportedOperationException(); // (3)
        }
    };
}
```

Notes on Example 6-22 on page 120:

- ▶ Again, we use an abstract implementation provided by the collections framework. The methods that we must implement are `iterator()` and `size()`.
- ▶ See Example 6-23 for the implementation of `VsamIterator`.
- ▶ Because there is no easy way to obtain the number of records in a VSAM data set (other than counting), we chose not to support this operation.

The hardest part is implementing an iterator class, as shown in Example 6-23. It is implemented as an inner class of `VsamMap` and so has access to `VsamMap`'s attributes.

Example 6-23 VsamMap.VsamIterator

```
public class VsamIterator implements Iterator {
    private final KeyedFileBrowse fb;
    private final KeyHolder kh = new KeyHolder();
    private byte[] nextKey;
    private byte[] nextValue;
    private final RecordHolder rh = new RecordHolder();

    public VsamIterator() {
        try {
            fb = ksds.startBrowse(new byte[keylength]); // (1)
        } catch (CicsException e) {
            throw new RuntimeException(e);
        }
    }

    public boolean hasNext() {
        if (nextKey == null) { // (2)
            try {
                fb.next(rh, kh); // (3)
                nextKey = kh.value;
                nextValue = rh.value;
            } catch (EndOfFileException e) {
                close(); // (4)
            } catch (CicsException e) {
                close(); // (5)
                throw new RuntimeException(e);
            }
        }
        return nextKey != null; // (6)
    }

    public Object next() {
        if (hasNext()) { // (7)
            Entry entry = new SimpleEntry(keyFromBytes(nextKey), valueFromBytes(nextValue));
            nextKey = nextValue = null;
            return entry;
        } else {
            throw new NoSuchElementException();
        }
    }

    public void remove() {
        throw new UnsupportedOperationException(); // (8)
    }

    public void close() {
        try {
```

```

        fb.end();
    } catch (CicsException ignored) {
    }
}
}

```

Notes on Example 6-23 on page 121:

- ▶ In CICS terms, to move through a data set sequentially, you start a BROWSE operation. In JCICS, this is implemented by the startBrowse() method, which returns an instance of class KeyedFileBrowse. This class, in turn, has methods to return the next record from the file.
 - ▶ To protect against multiple invocations of hasNext() without calling next() in between, we first check if the next record was already read. If not, we read it, using the next() method on the KeyedFileBrowse object we received in the constructor call. We remember both key and value.
 - ▶ If the end of file was reached, end the browse.
 - ▶ If any other exception occurred, end the browse to free any resources held by it, and throw a RuntimeException.
 - ▶ At this point, either the next record was successfully read, and nextKey (and nextValue) are non-null, or the end of file was reached, in which case nextKey remained null.
 - ▶ The implementation of next() simply calls hasNext(). If there was another record, construct a key/value pair from it, and reset nextKey and nextValue to null so that hasNext() tries a physical read at the next invocation.
 - ▶ There is no KeyedFileBrowse method to delete the current record, so we do not support this operation. It would be fairly easy to implement, however, by remembering the last key retrieved and calling KSDS.delete() with that key.
5. This method ends the browse, cleaning up any resources held by it.

Testing the implementation

To test our VSAM-backed map, we run a small program, shown in Example 6-24.

Example 6-24 Testing the VsamMap implementation

```

package com.ibm.itso.sg245275.vsam;

import java.util.Iterator;

public class VsamMapTest {
    private final VsamMap map;

    public static void main(String[] args) {
        try {
            new VsamMapTest();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private VsamMapTest() {
        String key = "66666";
        String record = createRecord(key);
        this.map = new VsamMap("ACCTFILE", key.length());
        System.out.println("Contains key " + key + ": " + map.containsKey(key));
        testPutNew(key, record);
    }
}

```

```

        System.out.println("Contains key " + key + ": " + map.containsKey(key));
        testPutExisting(key, record);
        testRemove(key);
        printAllValues();
    }

    private void testPutNew(String key, String record) {
        System.out.println("Adding record " + key);
        map.put(key, record);
    }

    private void testPutExisting(String key, String record) {
        try {
            System.out.print("Trying to modify record with key " + key + "..."); // (1)
            map.put(key, record.toString());
            System.out.println("Oops, should have caused an exception");
        } catch (IllegalArgumentException expected) {
            System.out.println("Caught an IllegalArgumentException, as expected");
        }
    }

    private void testRemove(String key) {
        System.out.println("Removing record " + key);
        map.remove(key);
    }

    private String createRecord(String key) {
        StringBuffer buf = new StringBuffer(key + "DOE                JOHN        "); // (2)
        buf.setLength(383); // Pad to record length
        return buf.toString();
    }

    private void printAllValues() {
        Iterator iter = map.values().iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next().toString().substring(0, 35)); // (3)
        }
    }
}

```

Notes on Example 6-24 on page 122:

- Modifying an existing record using `VsamMap.put()` is not allowed. Check that an `IllegalArgumentException` is thrown, as expected, which creates a record of the correct length for our example VSAM file, whose record length is 383 bytes. (The rest is padded with zeros.)
- In each iteration, we print the first 35 bytes of the record.

Note: There is one potential flaw in our implementation: After you start iterating over the map, there is no way to end the iteration prematurely (to end the BROWSE operation, in CICS parlance), which can potentially tie up resources.

In particular, the number of concurrent operations that can be processed against a VSAM file is limited by the value of the `STRINGS` attribute on the file definition (see *CICS Resource Definition Guide*, SC34-6228, for more information). So, when you open one iterator, and then at a later point open another one before the first one was at end-of-file, your program might block forever.

6.12 Interval control

An important facility in CICS is the ability to have one transaction start another transaction. You can have CICS start the transaction immediately (the default), at a specific time, or after a period of time. You can start the transaction on behalf of a specified terminal, and you can pass the started transaction some data. Additionally, the transaction can be started in another CICS region. The started transaction is asynchronous to the transaction that starts it. Thus, the started transaction is in its own unit of work.

Optionally, you can pass data to the started transaction. Other than being able to retrieve the data passed to it, the started transaction is independent of the originating transaction.

We used this facility when we started the HelloWorld sample program using the START command from CECI. To do it from a JCICS program, you set up an instance of class StartRequest and call the issue() method.

Optionally, you can pass data to the transaction being started, similar to the COMMAREA mechanism explained in 6.2.1, “Program control” on page 91. The difference is that because the starting transaction and the new transaction being started are independent from each other, the new transaction cannot pass data back to the originating transaction. Also, the mechanism to access the passed data is different: You use the Task.retrieve() method. Example 6-25 shows how it is done.

Example 6-25 Retrieving data passed from a START request

```
package com.ibm.itso.sg245275;

import java.util.BitSet;
import com.ibm.cics.server.*;

public class RetrieveDemo {

    public static void main(CommAreaHolder cah) {
        RetrievedDataHolder rdh = new RetrievedDataHolder();
        BitSet whatToRetrieve = new BitSet();                                // (1)
        whatToRetrieve.set(RetrieveBits.DATA);
        whatToRetrieve.set(RetrieveBits.QUEUE);
        try {
            Task.getTask().retrieve(whatToRetrieve, rdh);
            System.out.println("Retrieved DATA: " + new String(rdh.value.data)); // (2)
            System.out.println("Retrieved QUEUE: " + new String(rdh.value.queue)); // (3)
        } catch (EndOfDataException e) {                                     // (4)
            System.err.println("No data to retrieve. Did you start me from a terminal?");
        } catch (InvalidRetrieveOptionException e) {                       // (5)
            System.err.println("Expected option not set by START command");
        } catch (CicsException e) {
            e.printStackTrace();
        }
    }
}
```

Notes on Example 6-25:

- ▶ We must indicate what to retrieve: Data, an RTRANSID, an RTERMID, a queue name, or some combination of these. (See *CICS Application Programming Reference*, SC34-6232, for more information about the meaning of the various options.)
- ▶ To indicate what combination to retrieve, we must instantiate a BitSet and set the corresponding bits (one of the constants in the RetrieveBits interface).

- Print the data and the queue name we received.
- No data was passed.
- We tried to retrieve a piece of information that was not present. That is, either no data or no queue name was passed from the transaction that started ours.

You can test the sample using CEMT, as we show in Example 6-26.

Example 6-26 CEDA dialog to test Example 6-25 on page 124

```

START TR(SMP1) From('Your data goes here') Queue(MYQUEUE)
STATUS: ABOUT TO EXECUTE COMMAND                                NAME=
EXEC CICS START
  TRansid( 'SMP1' )
  < Interval( +0000000 ) | TIme() | ( After | AT ) < Hours() > < Minutes() >
    < SEconds() > >
  < FRom( 'Your data goes here' ) < Length( +00019 ) < FMh > > >
  < TErmid() | Userid() >
  < SYsid() >
  < RTRansid() >
  < RTErmid() >
  < Queue( 'MYQUEUE ' ) >
  < Nocheck >
  < Protect >
  < REqid() >
  < ATTach >
  < BRExit() < BRDATA() < BRDATALength() > > >

```

```

PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF

```

You enter the data in the From clause of the START command (you must use quotes if the data contains embedded blanks) and the queue name in the Queue clause. Try and omit one or both of them, and inspect the program's output.

6.13 Terminal services

JCICS terminal services allow interaction with the user terminal. You can send data to and receive data from the terminal, and send cursor and device control instructions to the terminal.

Alas, using these services is not for the faint of heart because they require intimate knowledge of the 3270 datastream format (see *3270 Data Stream Programmer's Reference*, GA23-0059).

There is one terminal service, however, that is relatively straightforward to use. If a JCICS program is started from a terminal, the Task.out variable (an instance of PrintWriter) allows you to write to that terminal, which is much like the standard Java System.out stream:

```
Task.getTask().out.println("Hello World!");
```

That said, there is good news. During the residency, we developed a little package boldly called JBMS (Java Basic Mapping Support), which provides an easy-to-use interface to the 3270 terminal, which is much like the corresponding BMS service in traditional CICS programming. Be aware, however, that the usual IBM Redbooks publication disclaimer applies: It is experimental code, not thoroughly tested, and certainly not ready for production purposes. It can come in handy, however, if you want to develop a simple front-end for an application without going through all of the effort to create a full-fledged Web interface.

We only show some particularly interesting parts of the JBMS package in the remainder of this section.

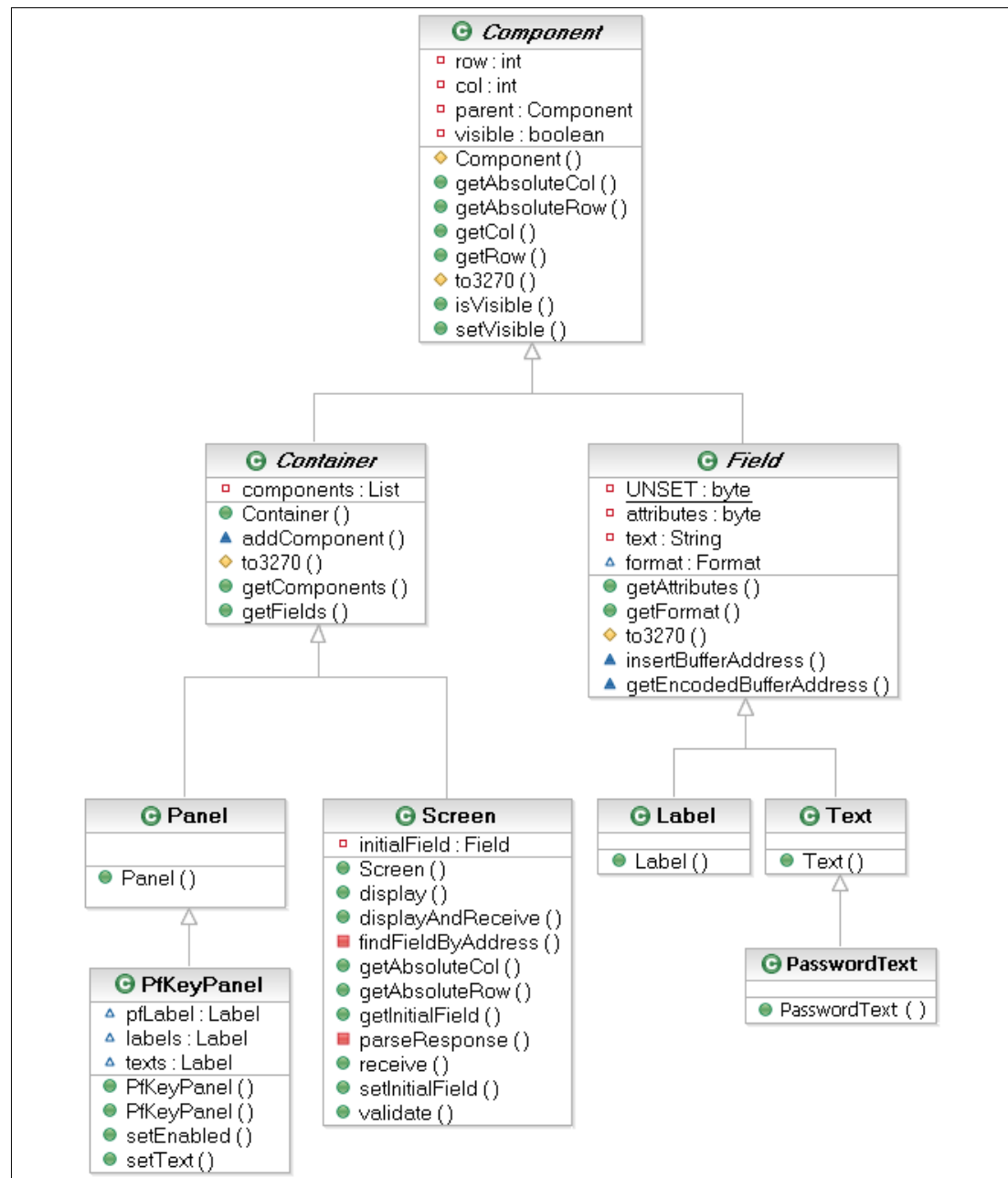


Figure 6-8 JBMS class hierarchy

A brief explanation of all classes follows.

Component

This is the abstract superclass for all classes that represent elements on the window. All components except *Screen* have a parent *Component* that must be passed to the constructor. A *Component* has a row/column position that is relative to its enclosing *Component* (if any) and a flag that indicates whether the component is visible, that is, must be displayed on the window at all.

Container

This is an abstract superclass for all components that contain other components. The idea is that you add related items to a *container* using a relative position. So, if you later decide that you must move a container to another position on the window, you simply change the container's position rather than the position of each individual item in the container.

Panel

Right now, a *panel* does not have additional functionality over a container. The idea is that it can add visual elements, such as a border as an indication of grouping.

For convenience, we provide a subclass, `PfKeyPanel`, which you can use to show PF-key assignments.

Screen

This container represents an entire 3270 screen. It is responsible for generating and sending the outbound 3270 data stream to the terminal, and for receiving and parsing the inbound data stream.

Again, there is a convenience subclass, `MenuScreen`, which can be used to easily create a menu screen showing several items and an input field for the user's choice.

Field

This is the abstract superclass of all basic components of a screen, that is, the actual fields that make up the screen.

Label

Represents fields that are intended for display only, that is, that are not for data entry by the user. However, the text displayed need not be static but can also be changed during the dialogue with the user, for example, to show an error message.

Text

This is a field for data entry. Optionally, you can set a formatter, that is, an instance of `java.text.Format`, to validate and format data entered by the user.

Next, we show you a sample screen created using the JBMS package (Figure 6-9 on page 128) and the program that created it (Example 6-27 on page 128).

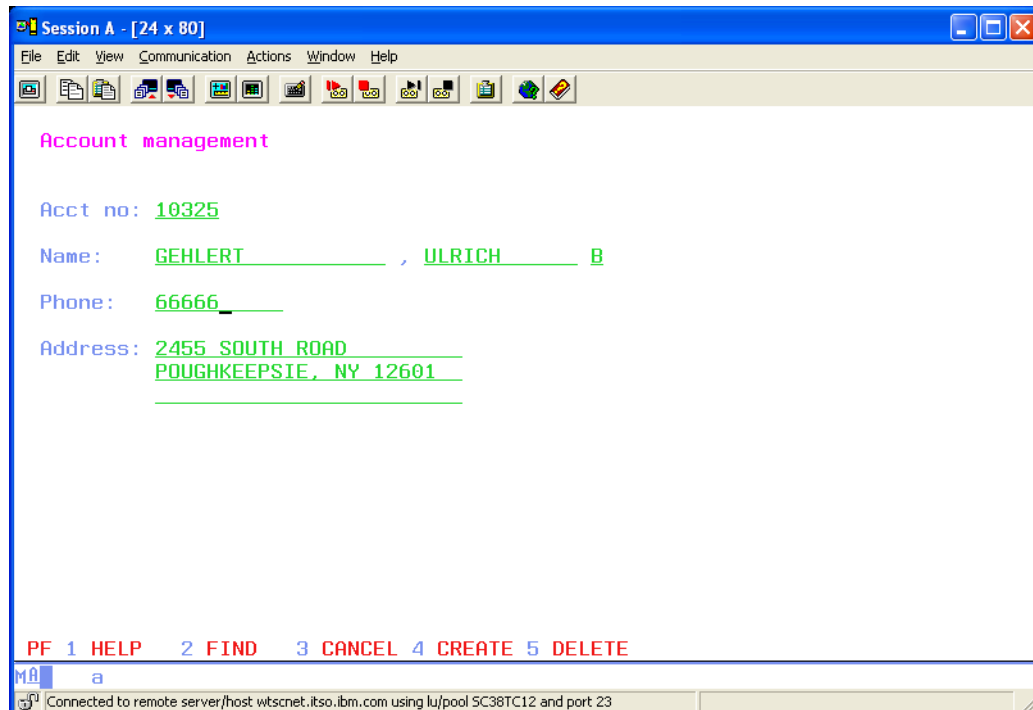


Figure 6-9 Sample screen produced by a JBMS program

Example 6-27 shows the program to generate this screen.

Example 6-27 Sample JBMS program

```
package com.ibm.itso.sg245275;

import java.text.DecimalFormat;

import com.ibm.cics.server.*;

import com.ibm.itso.sg245275.jbms.*;
import com.ibm.itso.sg245275.jbms.Field.Color;
import com.ibm.itso.sg245275.vsam.VsamMap;

public class BmsDemo {
    private Text acctno;
    private Text firstname, lastname, midinit;
    private Text phone;
    private Text[] addr = new Text[3];
    private PfKeyPanel pfkeys;
    private VsamMap accounts = new VsamMap("ACCTFILE", 5); // (1)

    public BmsDemo(TerminalPrincipalFacility term) throws InvalidRequestException,
        LengthErrorException, NotAllocatedException,
        TerminalException
    {
        Screen screen = createScreen();
        byte aid;
        String record = null;
        boolean end = false;

        term.clear();
        while (!end) {
```

```

        do {
            fillFields(record); // (2)
            aid = screen.displayAndReceive(term); // (3)
        } while (aid != AIDValue.PF3 && screen.validate() != null); // (4)
        switch (aid) {
            // ... omitted
        }
    }
    term.clear();
}

private String makeRecord() {
    // ... omitted
}

private void fillFields(String record) {
    // ... omitted
}

private Screen createScreen() { // (5)
    Screen screen = new Screen();
    Label label;

    label = new Label(screen, 1, 1, "Account management");
    label.setForegroundColor(Color.PINK);

    label = new Label(screen, 4, 1, "Acct no:");
    label.setForegroundColor(Color.TURQUOISE);
    acctno = new Text(screen, 4, 10, 5); // (6)
    acctno.setFormat(new DecimalFormat("00000"));

    // Name panel. // (7)
    Panel namePanel = new Panel(screen, 6, 0);
    label = new Label(namePanel, 0, 1, "Name: ");
    label.setForegroundColor(Color.TURQUOISE);
    lastname = new Text(namePanel, 0, 10, 18);
    label = new Label(namePanel, 0, 29, ",");
    label.setForegroundColor(Color.TURQUOISE);
    firstname = new Text(namePanel, 0, 31, 12);
    midinit = new Text(namePanel, 0, 44, 1);

    label = new Label(screen, 8, 1, "Phone: ");
    label.setForegroundColor(Color.TURQUOISE);
    phone = new Text(screen, 8, 10, 10);

    // Address panel.
    Panel addressPanel = new Panel(screen, 10, 0);
    label = new Label(addressPanel, 0, 1, "Address: ");
    label.setForegroundColor(Color.TURQUOISE);
    addr[0] = new Text(addressPanel, 0, 10, 24);
    addr[1] = new Text(addressPanel, 1, 10, 24);
    addr[2] = new Text(addressPanel, 2, 10, 24);

    // Create PF key bar.
    pfkeys = new PfKeyPanel(screen);
    pfkeys.setText(1, "HELP");
    // ... (omitted)

    // Initial cursor position.
    screen.setInitialField(acctno); // (8)
}

```

```

        return screen;
    }

    public static void main(CommAreaHolder ca) {
        // ... omitted
    }
}

```

Notes on Example 6-27 on page 128:

- ▶ The data displayed on this screen comes from a VSAM file. We use the `VsamMap` class developed in 6.11, “File control” on page 115, to access the data.
- ▶ Populate the fields with values from the record.
- ▶ Display the screen, and receive the user response. The fields that make up the screen are updated with the data that the user enters, and we get back an AID value that tells us what key the user pressed to send the data (for example, the ENTER key, or a PF key).
- ▶ If the user entered invalid data (in this example, non-numeric data in the acctname field), start all over again, unless he pressed PF3 for exit.
- ▶ This method creates the screen, adding labels for informative text, and fields for data entry.
- ▶ This is the account number field. We set up a formatter that causes the data to be displayed with leading zeros and validates the user input.
- ▶ For easier maintenance, we group related fields, such as the name and address fields together, with offsets relative to the enclosing container. To move a group of related fields, only change the position of the container rather than of each contained element.
- ▶ Set the initial field, that is, the one that the cursor is positioned on when the screen is displayed.

In the remainder of this section, we show you some implementation details. The most interesting parts are constructing the data to be sent to the 3270 terminal (in 3270 parlance, the *outbound data stream*), and parsing the data being sent back by the terminal (the *inbound data stream*).

Example 6-28 shows the method that creates the 3270 data stream and sends it to the terminal.

Example 6-28 Screen.display()

```

public void display(TerminalPrincipalFacility term) throws InvalidRequestException,
    LengthErrorException, NotAllocatedException, TerminalException
{
    ByteArrayOutputStream bos = new ByteArrayOutputStream();           // (1)
    bos.write(0);
    bos.write(0);
    to3270(bos);                                                     // (2)
    if (getInitialField() != null) {                                  // (3)
        // Position cursor to initial field.
        initialField.insertBufferAddress(bos, true);
        bos.write(Order.INSERT_CURSOR);
    }
    term.send(bos.toByteArray());                                     // (4)
}

```

Notes on Example 6-28 on page 130:

- ▶ Create a `ByteArrayOutputStream` to hold the outbound 3270 data stream.
- ▶ Create the 3270 data stream, as shown in Example 6-29.
- ▶ If an initial field was specified, write a 3270 instruction to set the current buffer position and another one to move the cursor to that position.
- ▶ This is the JCICS call to send the data stream to the terminal.

To generate the data stream, we recursively iterate over all components, as shown in Example 6-29.

Example 6-29 Container.to3270()

```
protected void to3270(ByteArrayOutputStream bos) {  
    if (isVisible()) {                                // (1)  
        for (Iterator iter = components.iterator(); iter.hasNext();) { // (2)  
            Component component = (Component) iter.next();  
            component.to3270(bos);                        // (3)  
        }  
    }  
}
```

Notes on Example 6-29:

- ▶ If this `Container` is visible at all:
 - Iterate over all contained components.
 - Tell each in turn to generate its part of the data stream, which is a recursive call if the current component is itself a container.

Finally, we must generate the 3270 stream for a single *field*, which is the most complicated part because here we finally must know about the 3270 data stream format.

Basically, the outbound 3270 data stream consists of command bytes (orders) followed by parameters. To define a field, use the Start Field (SF) or Start Field Extended (SFE) order. In our code, we use SFE.

A field begins at the current buffer position, which you can change with the Set Buffer Address (SBA) order and ends right before the next field begins. SBA is followed by a two-byte buffer address.

Therefore, to start a new field, we first set the current buffer position, and then define the field using SFE and finally create an empty *marker* field.

A 3270 field can have several attributes, such as color, intensity, and whether the field is protected (no data can be entered into the field). Each attribute is recognized by an attribute type byte followed by an attribute value byte (see Table 6-2 on page 132). One of those attributes, the 3270 attribute, is always present, and is stored in the 3270 display buffer at the first position of the field (it does not display on the screen). The other attributes are optional.

The SFE order is followed by a single byte that indicates how many type/value byte pairs follow.

Table 6-2 Some 3270 attributes and their possible values

Type code	Attribute type	Values	
0xC0	3270 Field attribute Value is a combination of bits. Some combinations have special meanings (autoskip, nondisplay). Value must be EBCDIC encoded.	0x20	Field is protected
		0x10	Field is numeric
		0x30	Field is autoskip (no entry)
		0x08	Intensified
		0x0C	Nondisplay (password)
0x41	Extended highlighting	0x00	Default
		0xF0	Normal (as determined by 3270 field attribute)
		0xF1	Blink
		0xF2	Reverse video
		0xF4	Underscore
0x42	Foreground color	0xF0	Neutral
		0xF1	Blue
		...	
		0xFF	White

Example 6-30 shows how the data stream for a single field is generated.

Example 6-30 *Field.to3270()*

```

protected void to3270(ByteArrayOutputStream bos) {
    if (isVisible()) {
        insertBufferAddress(bos, false); // (1)
        bos.write(Order.START_FIELD_EXT); // Start field extended // (2)
        bos.write(countExtendedAttributes()); // Number of extended attributes // (3)
        // Send extended attributes, as type/value byte pairs. // (4)
        writeAttribute(bos, AttributeType.STD3270, (byte) Util.encode(attributes));
        writeAttributeConditional(bos, AttributeType.EXT_HILITE, extHilite);
        writeAttributeConditional(bos, AttributeType.FOREGROUND, foregroundColor);
        try {
            bos.write(text.getBytes());
        } catch (IOException e) {
            // Can't happen with a ByteArrayOutputStream
        }
        // Send a dummy marker field to indicate end of current field // (5)
        bos.write(Order.START_FIELD);
        bos.write(Util.encode(Attribute.AUTOSKIP));
    }
}

```

Notes on Example 6-30:

- First we must tell the terminal the position of the field. The 3270 expects the position as a buffer address that is encoded in a rather peculiar format (see the full source code for details).
- Next, we indicate the start of a field using the Start Field Extended (SFE) order.

- SFE expects the number of attribute/value pairs.
- Write attribute/value pairs to the data stream.
- The 3270 assumes that a field ends where the next field starts, so we write a dummy marker field to indicate the end of the current one.

To get user input, we must receive and parse the inbound 3270 data stream, as shown in Example 6-31.

Example 6-31 Screen.receive() - Receive input from terminal, parse response, and update fields

```

/**
 * Receive input from terminal, and parse data stream.
 *
 * @return AID (Action Identifier) of the key pressed {one of the
 *         {@link com.ibm.cics.server.AIDValue AIDValue} constants).
 */
public byte receive(TerminalPrincipalFacility term) throws
    InvalidRequestException, LengthErrorException, NotAllocatedException, TerminalException
{
    DataHolder dah = new DataHolder();
    try {
        term.receive(dah);                                // (1)
    } catch (EndOfChainIndicatorException expected) {
        // Always thrown from TerminalPrincipalFacility.receive().
        // We can safely ignore it.                        // (2)
    }
    parseResponse(dah.value);                             // (3)
    return term.getAIDbyte();                             // (4)
}

```

Notes on Example 6-31:

- JCICS call to receive the response from the terminal.
- Term.receive() always throws an EndOfChainIndicatorException (because the underlying EXEC CICS API always sets the corresponding RESP condition), and we can safely ignore it.
- Parse the response and return the AID that caused the screen to be sent.

The inbound 3270 consists of Set Buffer Address (SBA) orders that indicate which field is being sent next, followed by the actual data (that is, whatever the user entered into the field). So, we inspect the data stream, looking for SBA orders and finding the corresponding JBMS Field object for each order, as shown in Example 6-32. Then we extract the field data and update the Field object accordingly.

Example 6-32 Parsing the response received from the 3270 terminal

```

private void parseResponse(byte[] resp) {
    // Start at index 1 (index 0 has a SET_BUFFER_ADDRESS for first field).
    for (int pos = 1; pos < resp.length; pos++) {           // (1)
        // Get buffer address.
        int bufaddr = ((resp[pos++] << 8) | (resp[pos++] & 0xFF)) & 0xFFFF; // (2)
        // Find end of current segment (look for next SBA).
        int end;
        for (end = pos; end < resp.length && resp[end] != Order.SET_BUFFER_ADDRESS; end++)
            ;
        // Find the field this segment corresponds to.
        Field field = findFieldByAddress(bufaddr);           // (3)
        if (field != null) {

```

```

        // Found field; set its contents to the data received.
        String fieldContents = new String(resp, pos, end - pos);           // (4)
        field.setText(fieldContents);
    }
    pos = end;
}
}

```

Notes on Example 6-29 on page 131:

- ▶ Scan the data stream, and examine it for SBA orders.
- ▶ Extract the buffer address (next two bytes after SBA).
- ▶ Look up the field that this buffer address refers to.
- ▶ Extract the field data, and set the new field text.

6.14 Using JZOS with CICS

The IBM JZOS Batch Toolkit for z/OS SDKs is a set of tools that address many of the functional and environmental shortcomings in current Java batch capabilities on z/OS. It includes a native launcher for running Java applications directly as batch jobs or started tasks, and a set of Java methods that make access to traditional z/OS data and key system services directly available from Java applications. Additional system services include console communication, multiline WTO (write to operator), and return code passing capability. In addition, JZOS provides facilities for flexible configuration of the run-time environment, and it allows intermediate data to be seen through z/OS System Display and Search Facility (SDSF). Java applications can be fully integrated as job steps to augment existing batch applications.

When working in CICS, use JCICS APIs in preference to JZOS to gain from the benefits offered by CICS, but there are times where JZOS might be useful to CICS Java developers. These include:

- ▶ When writing Java Batch applications that invoke CICS transactions using the J2C APIs
- ▶ Using the JZOS APIs to access PDS resources from a CICS region
- ▶ Converting COBOL and Assembler data type fields to Java objects

For more information about JZOS, see the *JZOS Installation and User's Guide*, SA23-2245-00 and the book *Java Stand-alone Applications on z/OS Volume II*, SG24-7291-00.



Evolving a heritage application using Java

In this chapter, we take a COBOL heritage application and describe ways that you can enhance it using the JCICS API. The stages of evolution are:

1. Implement a Web interface to coexist alongside the original 3270 display, which opens up the underlying business logic to a generation of Internet clients.
2. Fully implement the back-end business logic using Java, which introduces a pluggable data storage component for future scalability.
3. Implement a DB2 back-end, and migrate the application data from its existing VSAM setup.
4. Implement a Web service interface alongside the other interfaces into the program

You might find this section useful as both an approach for migrating heritage applications to JCICS and as a reference to example implementations of Web, file, and DB2 access using the JCICS API.

7.1 The heritage Trader application

The COBOL Trader sample is a 3270 application that allows you to buy and sell shares from a group of companies in real time. Written in COBOL, it has a CICS BMS interface and uses VSAM file access for data storage. The application is pseudo-conversational, which means that a chain of related non-conversational CICS transactions is used to convey the impression of a “conversation” to you as you go through a sequence of windows that constitute a business transaction. Figure 7-1 shows a breakdown of the heritage application.

The application consists of two modules:

- ▶ **TRADERPL:** The 3270 presentation logic. Invokes TRADERBL using an EXEC CICS LINK passing a COMMAREA structure for input and output.
- ▶ **TRADERBL:** The business logic. Data is persisted in VSAM files.

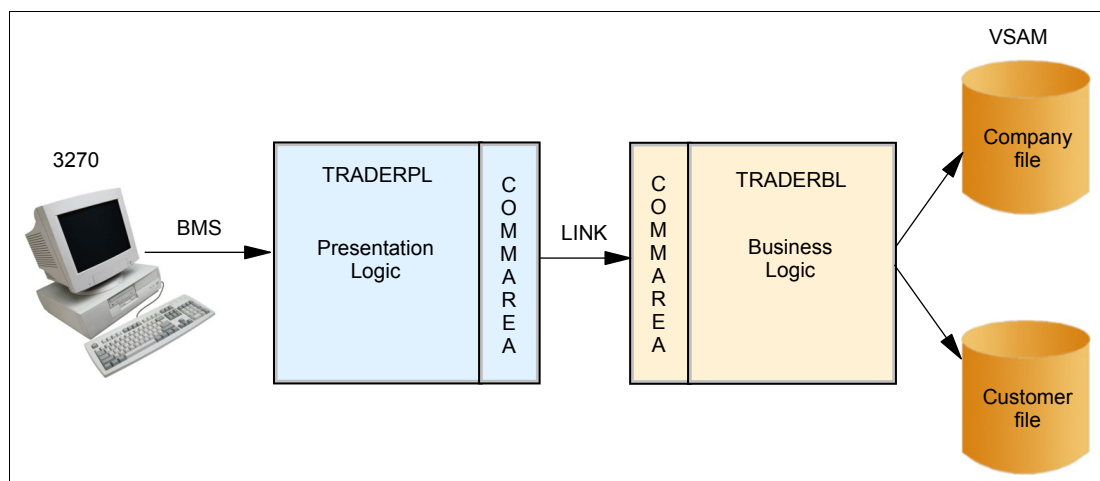


Figure 7-1 Breakdown of the heritage application

7.1.1 Installing the Trader application

The COBOL Trader application requires the following resources:

- ▶ Files
- ▶ Programs
- ▶ Mapset
- ▶ CICS definitions

Files

The Trader application uses two VSAM files:

- ▶ **COMPFILE:** Stores the list of companies and associated share prices. You can create it using the supplied JCL TRADERCOJCL.TXT, which requires as input the file TRADERCODATA.TXT.
- ▶ **CUSTFILE:** Stores the list of users and their share holding. You can create it using the supplied JCL TRADERCUJCL.TXT.

Programs

Two COBOL programs are used, TRADERPL and TRADERBL, which you must compile and put in a data set in your CICS region's DFHRPL concatenation. The programs are supplied as the files TRADERPL.TXT and TRADERBL.TXT, respectively.

Mapset

Trader uses the mapset NEWTRAD, which comprises the maps T001, T002, T003, T004, T005, and T006. The mapset is supplied in the file NEWTRAD.TXT, which you must assemble and put in your CICS region's DFHRPL concatenation.

CICS definitions

Define these CICS resources in the TRADER group.

- ▶ MAPSET(NEWTRAD)
- ▶ PROGRAM(TRADERBL)
LANGUAGE(COBOL)
- ▶ PROGRAM(TRADERPL)
LANGUAGE(COBOL)
- ▶ TRANSACTION(TRAN)
PROGRAM(TRADERPL)
- ▶ FILE(COMPFILE)
DSNAME(DATASET.CONTAINING.COMPFILE)
RLSACCESS(Yes)
STRINGS(5)
ADD(Yes)
BROWSE(Yes)
DELETE(Yes)
READ(Yes)
UPDATE(Yes)
- ▶ FILE(CUSTFILE)
DSNAME(DATASET.CONTAINING.CUSTFILE)
RLSACCESS(Yes)
STRINGS(5)
ADD(Yes)
BROWSE(Yes)
DELETE(Yes)
READ(Yes)
UPDATE(Yes)

For further information about creating the resource definitions for the Trader application, refer to the supplied file TRADERRDO.TXT, which contains the output of a CSD extract for group TRADER.

7.2 Other Extensions to the Trader application

There are other extensions to the Trader application that we wanted to try, but we did not have the time for. Two of these in particular are worthy of mention. They are both alternative ways of driving the Trader application, which we describe in the next sections.

7.2.1 Using WMQ Classes to drive the Trader application

It is useful to drive CICS transactions from messages using WebSphere MQ Series queues, which you can do easily with the MQ Series Java classes. All that is required is the following:

1. Define the application and transaction to CICS using the supplied CEDA transaction.

2. Ensure that the WebSphere MQ CICS adapter is installed in your CICS system. See WebSphere MQ for z/OS System Setup Guide for details.
3. Ensure that the JVM environment that is specified in CICS includes the appropriate CLASSPATH and LIBPATH entries.
4. Initiate the transaction from WebSphere MQ

More information about this and the CICS MQ adapter in general are in the CICS Transaction Server documentation. From CICS TS 3., 2 the MQ Adapter is owned and shipped by CICS.

7.2.2 Using the CICS Common Client Interface (CCI)

CICS TS 3.2 ships with support for the Common Client Interface (CCI), which is defined by the J2EE Connector Architecture Specification, Version 1.0 (JCA). The CCI connector for CICS helps you to build Enterprise JavaBean (EJB) server components that expose existing CICS programs.

This feature is particularly useful if you want to access CICS programs from WebSphere.

The CICS CCI provides a simple way for the Java client to access CICS programs. The CICS CCI has much in common with the CCI that is provided with CICS Transaction Gateway (CICS TG) but has the added benefit that the CICS CCI runs in a CICS region. CCI programs that are written to run outside of CICS, such as those that use the CICS TG CCI, can easily be migrated to the CICS CCI.

More information about CCI and JCA is in the CICS TS documentation, and in the JCA specification, which is available at this Web site:

<http://www.java.sun.com/j2ee/download.html>

7.3 Adding a JCICS Web interface

The Trader application is comprised of a presentation component and a business component with requests that are transmitted between them using a COMMAREA. Given an understanding of the COMMAREA structure, you can write and run additional interfaces alongside, or instead of, the original 3270 display, which Figure 7-2 on page 139 illustrates.

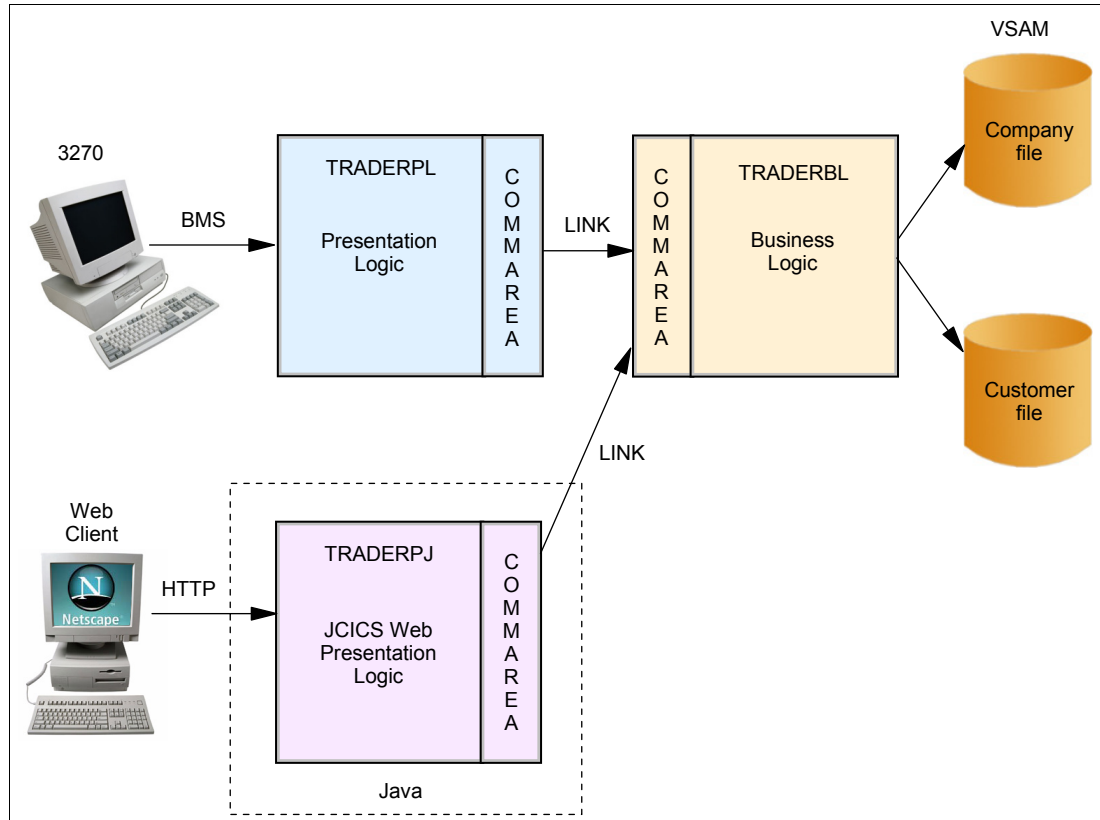


Figure 7-2 A Web interface alongside the heritage application

Following this approach, you implement a Web interface to the Trader application using the JCICS Web API.

7.3.1 Wrapping the COMMAREA

To transmit data requests to the COBOL business component, the JCICS program needs an understanding of the COMMAREA data. The COMMAREA is an array of bytes, the data of which is formatted according to a COBOL data structure. Given this information, you write a wrapper class in Java, which allows you to set and get the various values that are contained within the COMMAREA.

There are several methods that we can use to construct a wrapper class, and we introduce three of them in this book. In this section, we introduce:

- ▶ The first method in the next section, to construct a java class manually
- ▶ The second method in 7.3.2, “Wrapping the COMMAREA using JZOS” on page 142
- ▶ Third method in 7.3.3, “Wrapping the COMMAREA using J2C in RD/z” on page 144

Because the three methods generate different wrapper classes, we chose the first one for our Trader example. The biggest difference between the three generated classes are the function names in the class. It is easy to change from one method to another.

Example

As a starting point, we discuss wrapping the first two lines of the COMMAREA, as shown in Example 7-1.

Example 7-1 The first two lines of the COMMAREA from TRADERBL

```
01 COMMAREA-BUFFER.  
03 REQUEST-TYPE          PIC X(15).
```

The structure of the wrapper is a Java class that contains:

- ▶ A byte[] for the COMMAREA data
- ▶ A getter and setter method for the byte[]
- ▶ Variables that hold the index and length of each value in the COMMAREA
- ▶ Getter and setter methods for each value
- ▶ A utility method for formatting values

Example 7-2 The structure of the wrapper class

```
public class CommareaWrapper {  
  
    // Values in the COMMAREA  
    private final int[] COMMAREA_BUFFER = {0, 372};           // (1)  
    private final int[] REQUEST_TYPE = {0, 15};  
  
    // Variable to hold the binary data in the COMMAREA  
    private byte[] fData;  
  
    /* Instantiate the wrapper object and initialize its values */  
    public CommareaWrapper() {  
        fData = new byte[COMMAREA_BUFFER[1]];                // (2)  
        setNewValue("", 0, fData.length);  
    }  
  
    /* Set the binary data of the COMMAREA */  
    public void setByteArray(byte[] data) { fData = data; }  
  
    /* Get the binary data of the COMMAREA */  
    public byte[] getByteArray() { return fData; }  
  
    /* Get the value of REQUEST-TYPE */  
    public String getRequestType() {  
        return new String(fData, REQUEST_TYPE[0], REQUEST_TYPE[1]);  
    }  
  
    /* Set the value of REQUEST-TYPE */  
    public void setRequestType(String newValue) {  
        setNewValue(newValue, REQUEST_TYPE[0], REQUEST_TYPE[1]);  
    }  
  
    /*  
     * Inserts a value into the main byte[] and pads with spaces if necessary.  
     * Note: values will be left justified  
     */  
    private void setNewValue(String newValue, int index, int length) { ... }  
}
```

Notes on Example 7-2 on page 140:

- ▶ {0, 372} implies index=0 & length=372.
- ▶ Resets all values in the array to the space character.

You have a fully usable wrapper. To add further COMMAREA values, create a new variable to hold its index and length settings and the corresponding getter and setter methods. Generally the index is set to the index of the previous value plus its size, but sometimes it can also overlay a previous value depending on the COMMAREA structure.

Character-based values

The code so far applies to character-based *non-numeric* values. For character-based *numeric* values, you must implement an alternative `setNewValue()`, which right justifies and pads with zeros if required.

Important: COBOL character types can be flexible, so pay close attention during the creation of the wrapper. Feel free to modify and create new versions of the `setNewValue()` method as required.

Binary-based values

COBOL types that are binary based (COMP1, COMP3, and so on) need further processing.

The suggested approach is to write a getter method that converts the binary data into a Java primitive number (short, int, long) and gives it to the user. The setter method does the opposite and takes a Java primitive number, which it converts to the corresponding COBOL binary representation and inserts it into the `byte[]`.

Arrays

The COBOL statement `OCCURS X TIMES` refers to an array structure of size X. To allow for values that exist in arrays, the getter and setter methods must take an extra parameter, the index in the array:

- ▶ `public String getValue(int arrayIndex)`
- ▶ `public void setValue(String newValue, int arrayIndex)`

As shown in Example 7-3, the code for finding the index of a value in the array then becomes

`index = value-index + (value-length * array-index),`

Example 7-3 getter and setter methods for an array value

```
/* Get the value of ARRAY_VAL. Note: arrayIndex starts at 0 */
public String getArrayVal(int arrayIndex) {
    return new String(fData, ARRAY_VAL[0] + (ARRAY_VAL[1] * arrayIndex), ARRAY_VAL[1]);
}

/* Set the value of ARRAY_VAL. Note: arrayIndex starts at 0 */
public void setCompanyNameTab(String newValue, int arrayIndex) {
    setNewValue(newValue, ARRAY_VAL[0] + (ARRAY_VAL[1] * arrayIndex), ARRAY_VAL[1]);
}
```

Wrapping COBOL arrays: COBOL arrays can get quite complex, but it is still possible to wrapper them using indexing techniques similar to Example 7-3.

7.3.2 Wrapping the COMMAREA using JZOS

From version 2.2.1, JZOS supports automatic generation of record classes from COBOL copybooks and Assembler DSECTs. We use this function to generate our wrapper class.

Installing JZOS to z/OS

To install JZOS to z/OS:

1. Download the latest version JZOS package from the IBM AlphaWorks Web site.
2. Install it on our z/OS image. See the *JZOS Batch Launcher and Toolkit Installation and User's Guide*, SA23-2245-00.

After installation the following directories are created.

Name	Location
JZOS_HOME	/u/cicsrs5/jzos
Sample	CICSR5.JZOS.SAMPJCL
Loadlib	CICSR5.JZOS.LOADLIB

During installation, one directory is created in zFS. This directory contains the DLL and jars that are needed for JZOS, we specify this directory in <JZOS_HOME>

Changing JVM profile for JZOS

To change the JVM profile for JZOS:

1. Add <JZOS_HOME> to our LIBPATH_SUFFIX in JVM profile.
2. Add <JZOS_HOME>/ibmjzos.jar to our CLASSPATH_SUFFIX in JVM profile.

Example 7-4 shows the JVM profile for JZOS.

Example 7-4 JVM profile for JZOS

```
LIBPATH_SUFFIX=/u/cicsrs5/jzos:\nCLASSPATH_SUFFIX=/u/cicsrs5/jzos/ibmjzos.jar:\n
```

Generating the wrapper class

To generate the wrapper class:

1. Copy COMMAREA COBOL copybook to Sample directory. There is sample JCL named COBGEN that is used to generate record classes from COBOL copybooks.
2. Edit COBGEN following the guideline in comments.
3. Tailor the procedure and job for your installation:
 - a. Modify the Job card per your installation's requirements.
 - b. Customize the JCL SET variables.
 - c. Edit JAVA_HOME to point to the location of the Java SDK.
 - d. Edit JZOSAW_HOME to point to the JZOS alphaWorks® directory.
 - e. Modify MAINARGS DD arguments to RecordClassGenerator, as shown in Example 7-5 on page 143.

Example 7-5 MAINARGS DD

```
//MAINARGS DD *
com.ibm.jzos.recordgen.cobol.RecordClassGenerator
  bufoffset=false
  package=com.cobol.records
  outputDir=/u/cicsrs5
```

Notes about Example 7-5:

- Package is the package name of the generated java class.
- outputDir is the directory where we want to put the generated java class.

Details about how to use this JCL are in the JZOS Cobol Record Generator Users Guide.

4. Submit the JCL, which produces a Java class based on the provided COMMAREA.

Updating generated wrapper class

The generated wrapper class does not have the method `setByteBuffer(byte[] buffer)`, so we must manually add it.

Add `setByteBuffer()`, and change all of the `getxxx()` functions, as shown in Example 7-6.

Example 7-6 `setByteBuffer()` and `getxxx()`

```
public void setByteBuffer(byte[] buffer) {
    this._byteBuffer = buffer;
}
/*old getxxx function
    public String getRequestType() {
        if (requestType == null) {
            requestType = REQUEST_TYPE.getString(_byteBuffer);
        }
        return requestType;
    }
*/
public String getRequestType() {
    requestType = REQUEST_TYPE.getString(_byteBuffer);
    return requestType;
}
```

Now we have our wrapper class. It is much easier to use this method than to manually write a class because we do not need to know what data types are used in the copybook. JZOS manages all of them for us.

This is a manual process because we must update the generated java class and add the JZOS library and class to our JVM profile. Let us try another method of generating the wrapper class, J2C in RD/z.

7.3.3 Wrapping the COMMAREA using J2C in RD/z

The full name of J2C is J2EE Connector, which is a tool that you can use to create J2EE Connector artifacts that you can use to create enterprise applications. It helps you create a class or set of classes that map to COBOL, C, or PL/I data structures. In our example, we only use it to create a java data binding class. J2C is part of RD/z and is available to all users.

CICS/IMS Java Data Binding wizard

The CICS/IMS Java Data Binding wizard guides you step-by-step to generate a wrapper class:

1. Switch to the J2EE perspective.
2. To start the J2C dynamic wizard, from the menu bar, select **File** → **New** → **Other** → **J2C**.
3. Select **CICS/IMS Java Data Binding**, as shown in Figure 7-3.

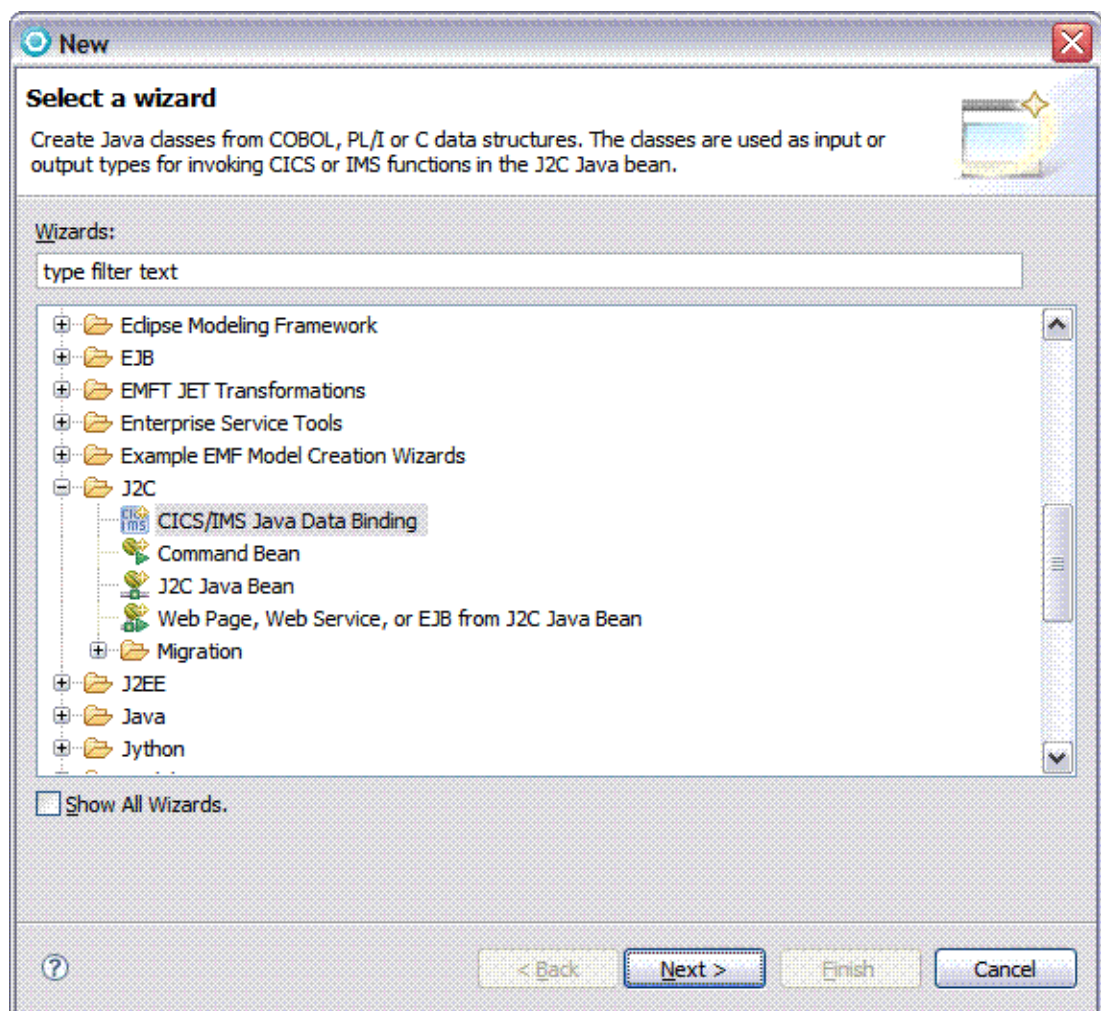


Figure 7-3 Select CICS/IMS Java Data Binding

4. Click **Next**.
5. On the Specify data import configuration properties page, specify the data types that your binding class uses and the location of the COBOL file, as shown in Figure 7-4 on page 145.

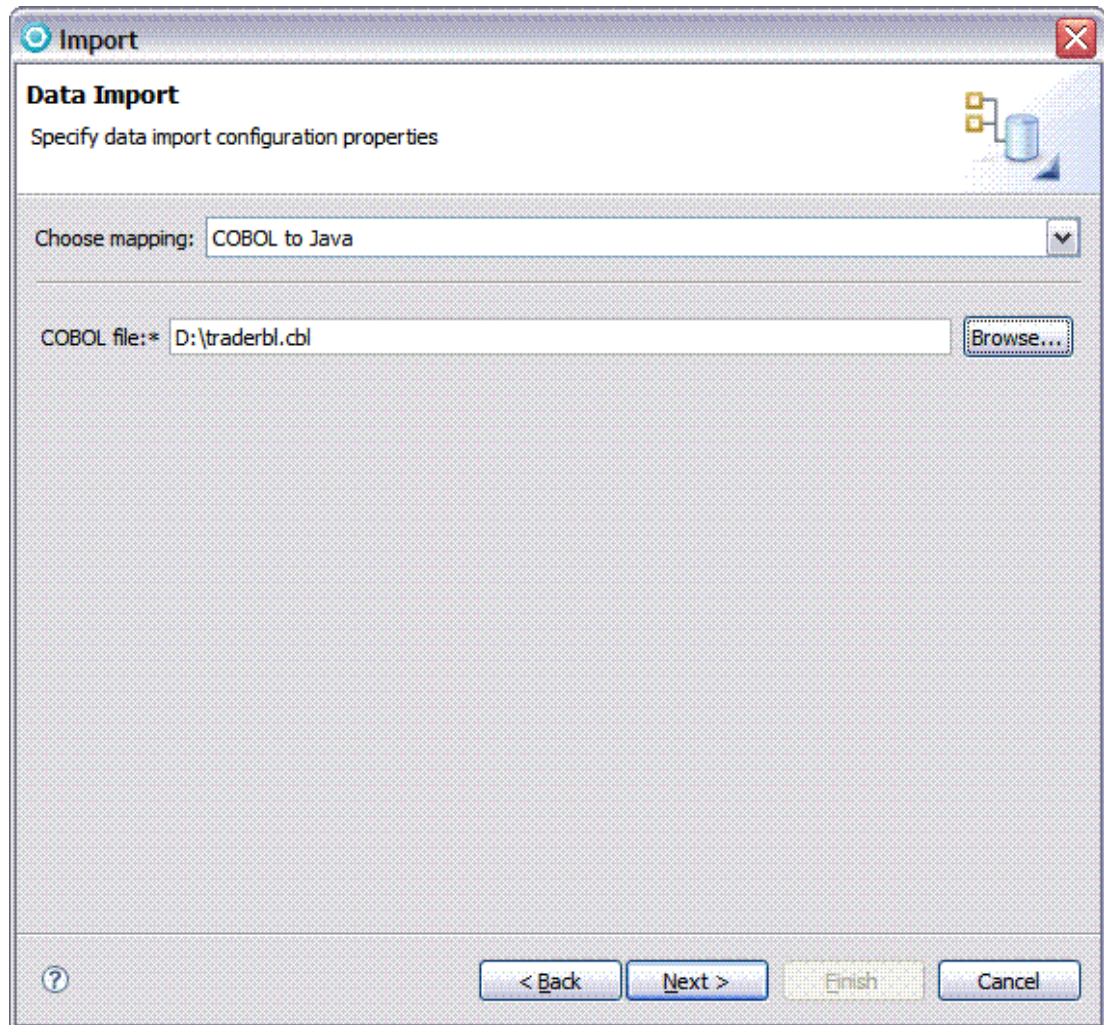


Figure 7-4 Select mapping type and specify COBOL file

Mapping types that you can use:

- ▶ COBOL to Java
- ▶ C to Java
- ▶ COBOL MPO to Java (For output data bindings only)
- ▶ C MPO to Java (For output data bindings only)
- ▶ PL/I to Java
- ▶ PL/I MPO to Java (For output data bindings only)
- ▶ J2C support COBOL, C and PL/I (We choose COBOL to Java here)

6. Click **Next**. Choose the platform and data structure that you want to map, as shown in Figure 7-5 on page 146.

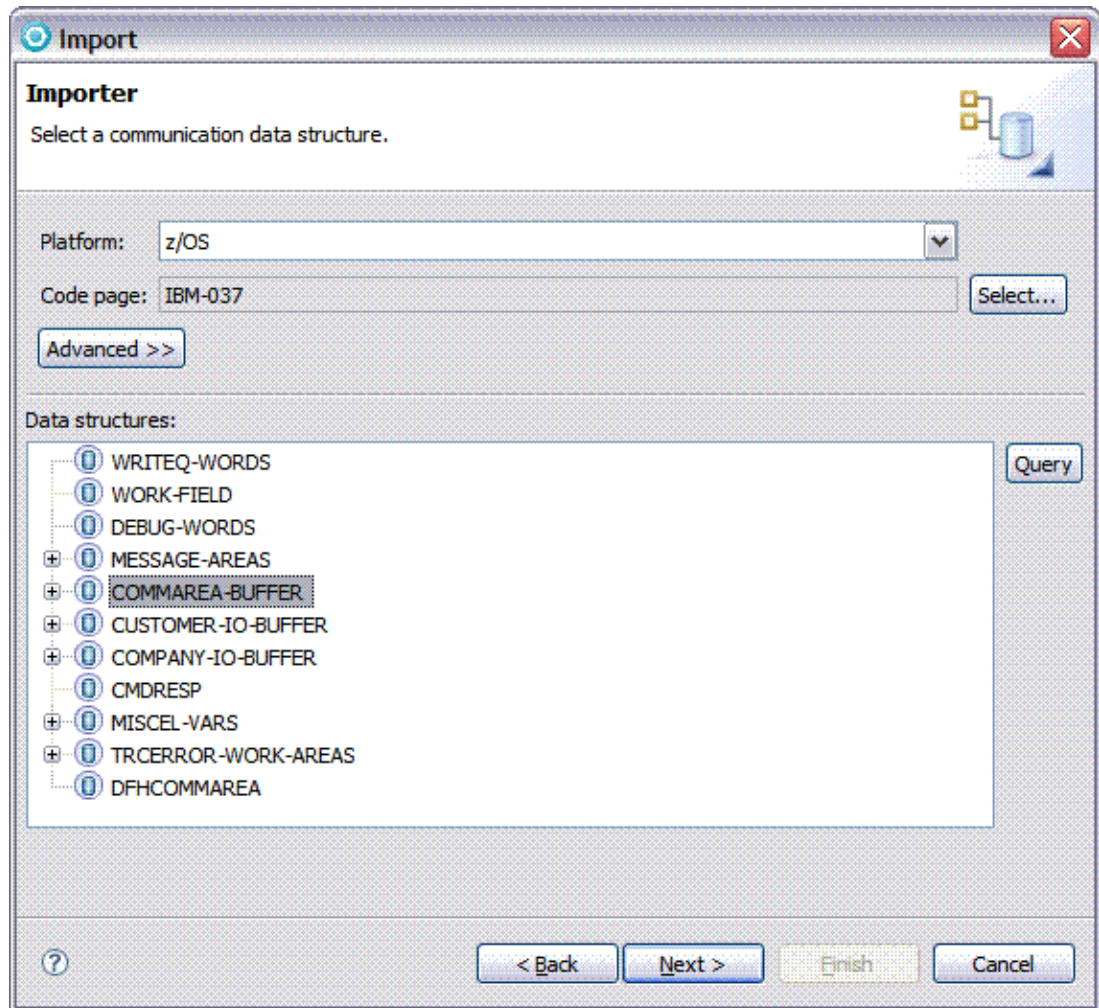


Figure 7-5 select platform and communication data structure

Important: Make sure that you choose z/OS platform here. the default value is win-32.

7. Click **Next**. Select the Project and Package Names, as shown in Figure 7-6 on page 147.

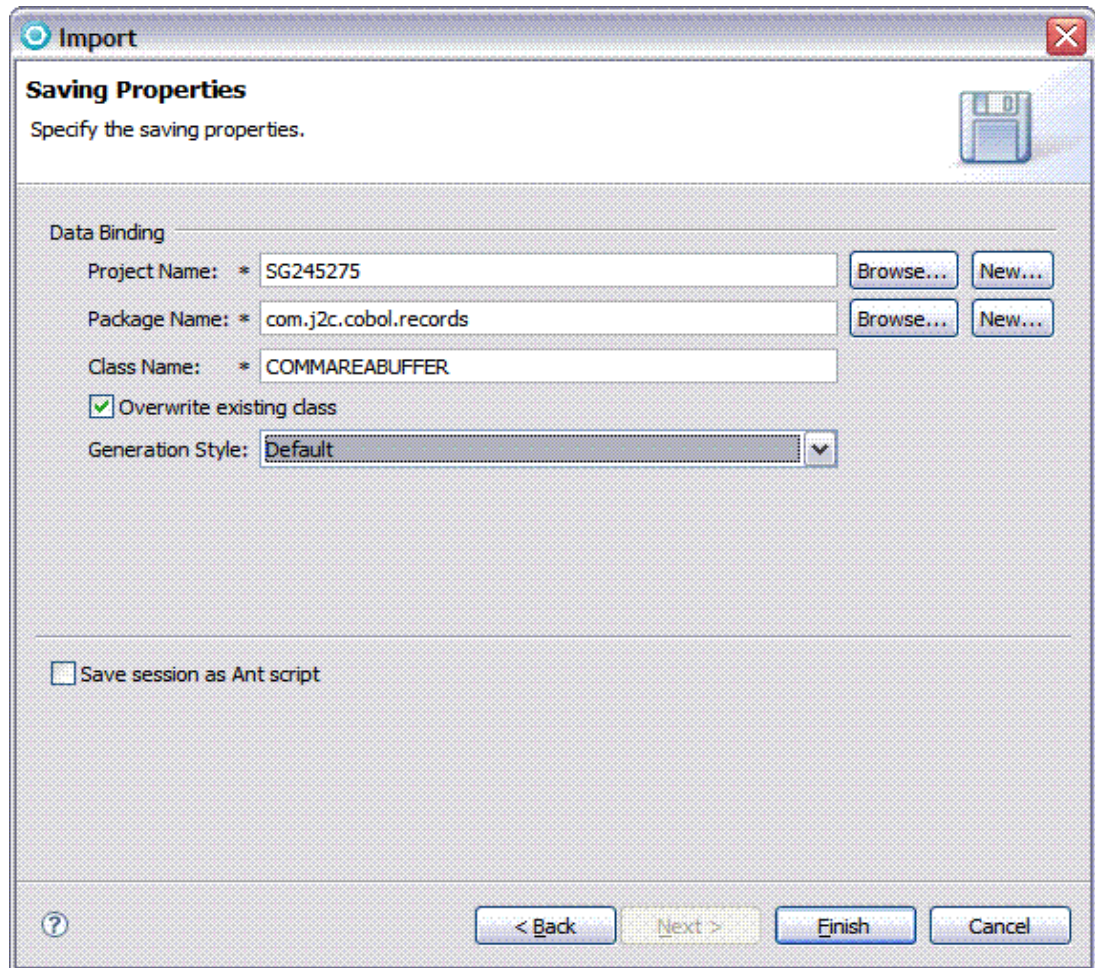


Figure 7-6 Select project name and package name

8. Click **Finish** to create the wrapper class `COMMAREABUFFER.java`.

Important: This wrapper class depends on `marshall.jar` and `J2EE.jar`, so you must upload these two jar to your zFS, and add them into the `CLASSPATH_SUFFIX` of JVM profile.

You can also generate java classes for accessing a VSAM file. To do this, return to Figure 7-5 on page 146, and select **CUSTOMER-IO-BUFFER** to generate a java class for accessing file `CUSTFILE`. Alternately, select **COMPARY-IO-BUFFER** to generate a java class for accessing file `COMPFILE`.

We created the required wrapper class, which includes all of the getter and setter methods, so no manual code is needed. A few button clicks and the class is generated. J2C is the easiest option of the three methods that are available.

7.3.4 Understanding COMMAREA request formats

Having written the CommareaWrapper, we now must understand how to format the requests to send to TRADERBL. Observing the link commands in TRADERPL, the requests can be broken down to the following settings in the COMMAREA structure:

- ▶ Get list of companies:
 - REQUEST-TYPE = "Get_Company"
- ▶ Get a company quote:
 - REQUEST-TYPE = "Share_Value"
 - RETURN-VALUE = "00"
 - USER-ID = User ID of current user
 - COMPANY-NAME = Company selected for previous menu
- ▶ Buy shares:
 - REQUEST-TYPE = "Buy_Sell"
 - RETURN-VALUE = "00"
 - USER-ID = User ID of current user
 - COMPANY-NAME = Company selected for previous menu
 - UPDATE-BUY-SELL = "1"
- ▶ Sell shares:
 - REQUEST-TYPE = "Buy_Sell"
 - RETURN-VALUE = "00"
 - USER-ID = User ID of current user
 - COMPANY-NAME = Company selected for previous menu
 - UPDATE-BUY-SELL = "2"

If you set these values in the CommareaWrapper and link to TRADERBL the data returned is exactly the same as that for TRADERPL.

7.3.5 A test Web application

To test the CommareaWrapper class, we now write a simple JCICS Web program to interface with the COBOL application TRADERBL. The program retrieves a list of companies and displays them in a Web page.

SimpleTraderPL

Using the JCICS Web API create a Java class called SimpleTraderPL. As with all JCICS programs, it needs a public static void main(CommAreaHolder commAreaHolder) method, which in this case gets called by the CICS Web transaction CWBA.

The flow of the program is:

1. Build the COMMAREA data.
2. Link to TRADERBL passing the COMMAREA.
3. Build the HTML from the returned COMMAREA data.
4. Send the HTTP response.

Building the COMMAREA is a matter of creating a CommareaWrapper object instance and setting the requestType value to "Get_Company", as shown in 7.3.4, "Understanding COMMAREA request formats" on page 148. To alleviate any potential errors that might occur if this value is referenced in multiple places, it is stored as a static variable GET_COMPANY_REQ in the class TraderConstants.

Next the request is sent using a CICS program link to TRADERBL. A call to `getByteArray()` extracts the `byte[]` from `CommareaWrapper` and passes it to the `JCICS link()` command. Any COMMAREA data that TRADERBL changes is reflected back in this `byte[]`.

Program names are case sensitive: Program names specified for the `JCICS link()` command are case sensitive and must be presented in uppercase.

If the link command fails for any reason, the exception data is outputted to Java's `stderr`. See 8.3.4, "JVM stdout and stderr" on page 206, for more information.

The program now has the list of companies and begins to build the HTML response. Dynamic HTML content is built using documents that allow a user to insert HTML line-by-line or in one big chunk. In `JCICS`, a document object is instantiated and HTML added using the `appendText()` method. In `SimpleTraderPL`, company names are extracted from the `COMPANY-NAME-TAB` array value in the `CommareaWrapper` and inserted into the HTML. Lastly, the document is sent back in the HTTP response. If anything fails, an attempt is made to send back a HTTP 500 error response.

HTTP response values: The parameters to the `HttpServletResponse.sendDocument()` method supply HTTP response values, such as status code, status response, and client code page.

Example 7-7 shows the syntax for the `SimpleTraderPLmain()` method.

Example 7-7 The SimpleTraderPL main() method

```
public static void main(CommAreaHolder commAreaHolder) {
    final CommareaWrapper commareaWrapper = new CommareaWrapper();

    // Prepare COMMAREA for request to COBOL program TRADERBL
    commareaWrapper.setRequestType(TraderConstants.GET_COMPANY_REQ);

    // Send the data request to TRADERBL
    final Program program = new Program();

    HttpServletResponse response = null;
    Document document = null;

    try {
        // Link to TRADERBL
        program.setName("TRADERBL");
        program.link(commareaWrapper.getByteArray());

        // Build the HTML response
        response = new HttpServletResponse();
        document = new Document();

        document.appendText("<H1>SimpleTraderPL</H1>");
        document.appendText("<P>List of companies:</P>");
        document.appendText("<UL>");
        document.appendText("<LI>" + commareaWrapper.getCompanyNameTab(0) + "</LI>");
        document.appendText("<LI>" + commareaWrapper.getCompanyNameTab(1) + "</LI>");
        document.appendText("<LI>" + commareaWrapper.getCompanyNameTab(2) + "</LI>");
        document.appendText("<LI>" + commareaWrapper.getCompanyNameTab(3) + "</LI>");
        document.appendText("</UL>");

        // Send the response
```

```

        response.sendDocument(document, (short) 200, "OK", "iso-8859-1"); // (1)
    }
    catch (Exception e) { sendErrorResponse(); e.printStackTrace(); }
}

private static void sendErrorResponse() {
    try {
        final Document doc = new Document();
        doc.appendFromTemplate("Internal Error");
        new HttpResponse().sendDocument(doc, (short) 500, "Internal Error", "iso-8859-1");
    }
    catch (Exception e) { System.err.println("Exception: " + e); e.printStackTrace(); }
}

```

Notes on Example 7-7 on page 149:

- ▶ 200 is the HTTP status code for OK.
- ▶ iso-8859-1 is the name of the Latin-1 (ascii) character set.

Further features of documents are used in “Displaying the next page” on page 156.

Installing the Trader into CICS

The source code and compiled classes for this example are in `Trader-1.jar`. Follow the instructions in Chapter 3, “Setting up CICS to run Java applications” on page 37, and make sure this jar file is added to the CICS Java class path.

You must define two new CICS resources alongside the existing definitions in the TRADER group:

- ▶ PROGRAM(TRADERPS)
 CONCURRENCY(Threadsafe)
 JVM(Yes)
 JVMCLASS(com.ibm.itso.sg245275.trader.SimpleTraderPL)
- ▶ TCPIPSERVICE(TCPIPS)
 URM(DFHWBADX)
 PORTNUMBER(<portnumber>)
 PROTOCOL(Http)
 TRANSACTION(CWXN)
 AUTHENTICATE(No)

For the TCPIPSERVICE definition, make sure that you choose a port number that is available and not already open. Also ensure the group DFHWEB is installed to pick up the necessary Web transactions.

Note: The DFHWEB group is in DFHLIST, so if you have this in your CICS startup, the Web transactions are installed by default.

Running SimpleTraderPL

To run SimpleTraderPL:

1. Open your favorite Web browser, and type in a URL similar to the following:
`http://wtsc66.itso.ibm.com:5275/cics/cwba/traderps`
2. Ensure that the port number matches up to that in your TCPIPSERVICE definition and that the program name at the end is `traderps`.

Note: The program name at the end of the URL is not case sensitive.

The browser now displays the list of companies, as shown in Figure 7-7.

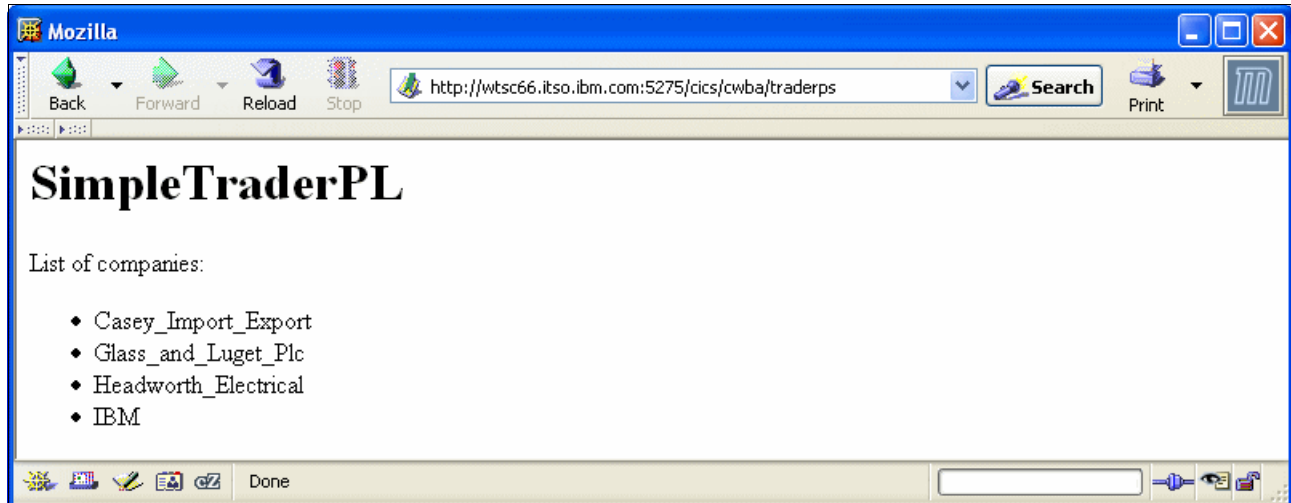


Figure 7-7 SimpleTraderPL's browser display

Congratulations. You implemented your first JCICS Web interface.

7.3.6 Designing the HTML interface

After successfully running the sample, you can begin developing JCICS Web applications. But before jumping in feet first with the coding, let us take a look at the Web site's navigation design.

The BMS display

The 3270 BMS display allowed users to traverse through menus while making navigation decisions along the way. From each menu, they can always get back to the previous one. A graphical representation of the navigation paths is shown in Figure 7-8 on page 152.

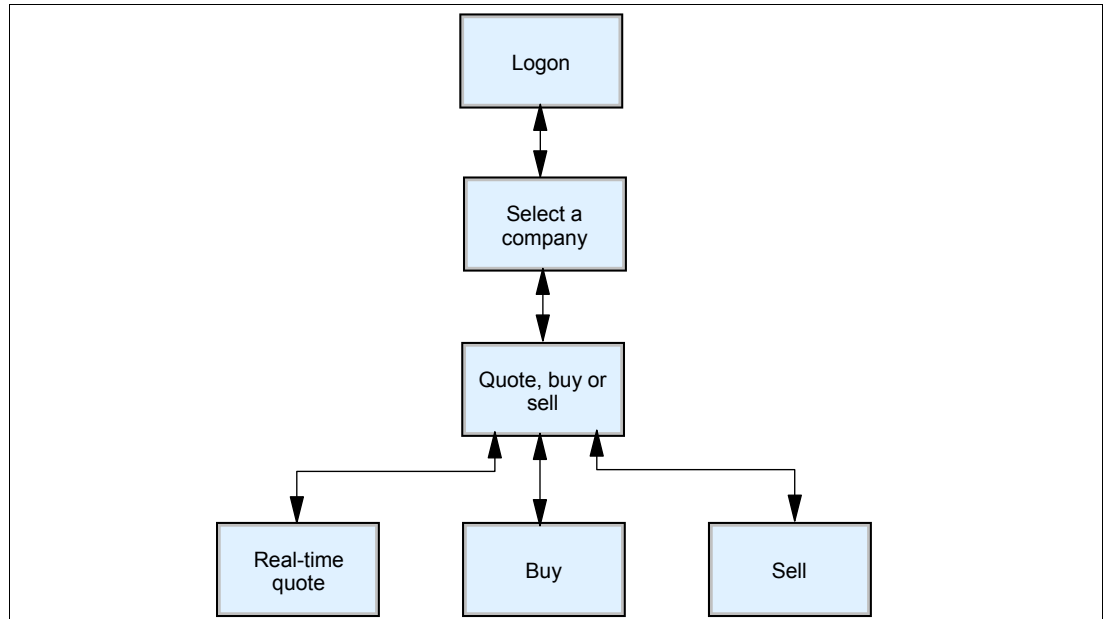


Figure 7-8 Navigation of the Trader application

The HTML design

The HTML pages navigation follows a similar design to the 3270 display, although there are a couple of changes that you can make to increase usability. An explicit login page is no longer needed because CICS has the option of handling Web security implicitly for you. For more information about Web security, see 7.3.9, “Web security” on page 160.

Utilizing the flexibility of HTML navigation, the company selection and share trading options pages are combined to provide a single interface for showing the companies and actions that you can perform on them. For further convenience the company quotes page contains direct links to the share buying and selling pages for that company. As a usability enhancement, the buy and sell pages also display some of the company information that is available from the company quotes page.

Figure 7-9 shows the new navigation design.

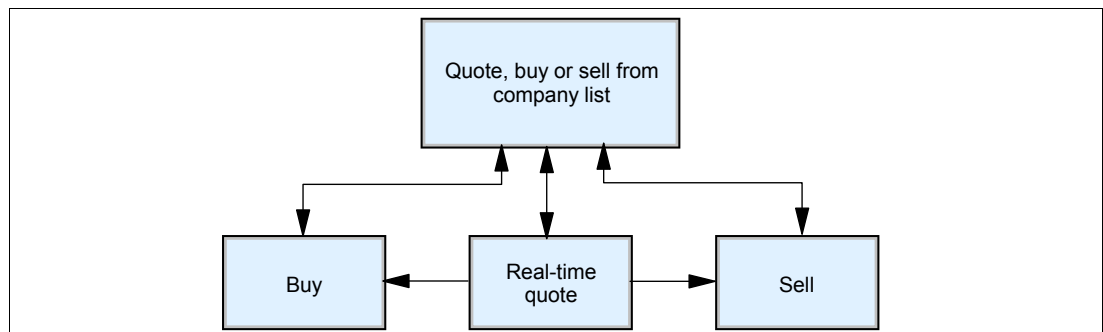


Figure 7-9 Web interface navigation

7.3.7 Implementing the design

With the design complete, you can now implement the code. For the purpose of maintainability, the HTML Web pages are stored in PDS members and accessed using the

CICS document's API, which is fully supported in JCICS. Documents allow you to build dynamic HTML using CICS templates and symbols.

Note: PDS members have a fixed width, so HTML files that are uploaded using FTP are checked to ensure that long lines are not truncated.

TraderPL

The JCICS Web program TraderPL is a single point of entry to all Web requests. Parameters received through HTTP GET and POST requests control the response of the program.

The main flow of logic through the program is:

1. Get HTTP input fields.
2. Perform buying and selling.
3. Get request details.
4. Display next page.

Getting HTTP input fields

For convenience, the program actions are stored as static strings. These values must match the form field values that are specified in the HTML. Example 7-8 shows action constants.

Example 7-8 Action constants

```
private static final String PERFORM_BUY  = "performbuy";
private static final String PERFORM_SELL = "performsell";
private static final String SHOW_LIST    = "showlist";
private static final String SHOW_DETAILS = "Details";           // (1)
private static final String SHOW_BUY     = "Buy";
private static final String SHOW_SELL    = "Sell";
private static final String LOGOFF       = "Logoff";
```

Note on Example 7-8:

The HTML equivalent for this is:

```
<INPUT type='submit' name='action' value='Details'>
```

If the action is to buy or sell shares, then after this is performed the user is redirected to the company details page. When no parameter is supplied, the default action is to show the company list page. Alternately, if an unknown action is specified, then a message is sent back to the user.

Example 7-9 shows the main flow of logic in TraderPL.

Example 7-9 The main flow of logic in TraderPL

```
private String fCompanyName;
private String fNumShares;
private String fMessage;

public TraderPL() {

    // Get HTTP data
    final HttpRequest request = HttpRequest.getHttpRequestInstance();

    String action = null;

    try {
        action      = request.getFormField("action");
```

```

        fCompanyName = request.getFormField("companyname");
        fNumShares   = request.getFormField("numshares"); // Used when buying/selling shares
    }
    catch (InvalidRequestException e) {}

    // If action is not set then default to show company list screen
    if (action == null) action = SHOW_LIST;
    // Buy/sell shares and then redirect to show company details screen
    else if (PERFORM_BUY.equals(action)) {
        performBuy();
        action = SHOW_DETAILS;
    }
    else if (PERFORM_SELL.equals(action)) {
        performSell();
        action = SHOW_DETAILS;
    }

    // Show desired page
    if (SHOW_LIST.equals(action))      showList();
    else if (SHOW_DETAILS.equals(action)) showDetails();
    else if (SHOW_BUY.equals(action))  showBuy();
    else if (SHOW_SELL.equals(action)) showSell();
    else if (LOGOFF.equals(action))    logoff();
    else {
        // Notify the user of the unknown action
        fMessage = "Unknown action: '" + action + "'"; // (1)

        // Invalid page requested so show company list page instead
        showList();
    }
}

```

Note on Example 7-9 on page 153:

- This message is output as part of the HTTP response.

Bypassing case-sensitive HTTP parameters: By default, HTTP parameters are case sensitive. To bypass this your Java program can use the `String.toLowerCase()` function for string comparisons.

Perform buy/sell

The `performBuy()` and `performSell()` methods are virtually identical. The only difference is the value for UPDATE-BUY-SELL in the COMMAREA, as shown in 7.3.4, “Understanding COMMAREA request formats” on page 148.

The method structure is:

1. Build request COMMAREA.
2. Send request.
3. Check return value.

A non-zero return value generates a message that is returned in the HTTP response. For convenience, we added static constants for the return values and their corresponding messages to the `TraderConstants` class, which we first mentioned in “SimpleTraderPL” on page 148. It also has a utility method for passing a return code and getting back its corresponding message.

Example 7-10 on page 155 contains the syntax for the `performBuy()` method.

Example 7-10 The performBuy() method

```
private void performBuy() {
    // Prepare commarea for request to COBOL program Traderbl
    fCommareaWrapper.setRequestType(TraderConstants.BUY_SELL_REQ);
    fCommareaWrapper.setReturnValue(TraderConstants.CLEAN_RETURN);
    fCommareaWrapper.setCompanyName(fCompanyName);
    fCommareaWrapper.setNoOfSharesDec(fNumShares);
    fCommareaWrapper.setUpdateBuySell(TraderConstants.SUBTYPE_BUY);

    // Send the data request to TRADERBL
    sendLinkRequest();

    // If the link failed then a message was set by sendLinkRequest.
    // If it worked then check the return value from TRADERBL
    if (fMessage == null)
        fMessage = TraderConstants.getMessage(fCommareaWrapper.getReturnValue());
}
```

Get request details

All of the show*() functions follow a common design:

1. Build request COMMAREA.
2. Send request.
3. Check return value.
4. Build symbol list.
5. Send HTTP response.

Get_Company: The Get_Company request to TRADERBL does not produce a return value and so is not checked.

The appropriate values are set in the CommareaWrapper and the request sent to TRADERBL. When the request returns, the symbol list is built using a series of name/value pairs (see *CICS Application Programming Guide*, SC34-6231, for more information about symbols). As you can see in Example 7-11, values are extracted from the CommareaWrapper and inserted into the symbol list. For any numerical values that are displayed on pages, such as the company quotes page, a custom utility method strip() is used to remove any leading zeros for the code of this method.

Example 7-11 The showList() method

```
private void showList() {
    // Prepare COMMAREA for request to COBOL program Traderbl
    fCommareaWrapper.setRequestType(TraderConstants.GET_COMPANY_REQ);

    // Send the data request to TRADERBL
    sendLinkRequest();

    // Return value is not checked as none are returned from a GET_COMPANY_REQ request

    // Set appropriate symbol values
    final String symbolList =
        "company1=" + fCommareaWrapper.getCompanyNameTab(0) + "&" +
        "company2=" + fCommareaWrapper.getCompanyNameTab(1) + "&" +
        "company3=" + fCommareaWrapper.getCompanyNameTab(2) + "&" +
        "company4=" + fCommareaWrapper.getCompanyNameTab(3) + "&" +
        "message=" + (fMessage != null ? fMessage : ""); // (1)

    // Send the response
```

```
        sendResponse(symbolList, "TRADCOML"); // (2)
    }
```

Notes on Example 7-11 on page 155:

- ▶ If a message was generated, its symbol value is set or else it is left blank.
- ▶ TRADCOML is a DOCTEMPLATE definition that needs to exist in CICS. For more information, see “Installing TraderPL” on page 157.

The `sendLinkRequest()` method, shown in Example 7-12, is identical to the code in “SimpleTraderPL” on page 148 other than a single addition. Most requests to TRADERBL require a user ID, and because there is no longer an explicit login page, as discussed in 7.3.6, “Designing the HTML interface” on page 151, the user ID is retrieved using the JCICS method `Task.getTask().getUserID()`.

Example 7-12 The `sendLinkRequest()` method

```
private void sendLinkRequest() {
    final Program program = new Program();

    program.setName("TRADERBL");

    try {
        // Set the UserId value in the COMMAREA
        fCommareaWrapper.setUserId(Task.getTask().getUserID());

        program.link(fCommareaWrapper.getByteArray());
    }
    catch (Exception e) { System.err.println(e); e.printStackTrace(); fMessage = "Error - "
+ e; }
}
```

Note: If security is disabled, the user ID defaults to CICSUSER. See 7.3.9, “Web security” on page 160 for enabling security in Web applications.

Displaying the next page

The `sendResponse()` method completes the HTTP request by building the document object, adding the symbol list to it, appending the corresponding DOCTEMPLATE, and sending the HTTP response. If anything goes wrong while setting up the document object, then the exception data is outputted to Java’s `stderr`, and an error message is sent back to the user in the HTTP response. An error occurs if the template it is trying to append was not installed in CICS.

Tip: Use CEMT INQUIRE DOCTEMPLATE to ensure your DOCTEMPLATE was installed.

As discussed in “SimpleTraderPL” on page 148, the same parameter options are used in the `HttpResponse.sendDocument()` method, which Example 7-13 shows.

Example 7-13 The `sendResponse()` method

```
private void sendResponse(String symbolList, String template){
    final HttpResponse response = new HttpResponse();

    Document document          = null;
    boolean appendTemplateWorked = false;

    try{
```

```

        // Create a new document and set the symbol list
        document = new Document();
        document.setSymbolList(new SymbolList(symbolList));           // (1)

        // Append the template
        // Note this needs to have been defined in cics as a DOCTEMPLATE resource
        document.appendTemplate(template);

        // If we get here then the append succeeded
        appendTemplateWorked = true;
    }
    catch (Exception e) { sendErrorResponse(); e.printStackTrace(); }

    // If the append failed then send back a message using the document object
    if (!appendTemplateWorked && document != null) {
        try {
            document.appendText("Problem loading template. See Java stderr for more info");
        }
        catch (Exception e) {}
    }

    try { response.sendDocument(document, (short) 200, "OK", "iso-8859-1"); }
    catch (Exception e) { sendErrorResponse(); e.printStackTrace(); } // (2)
}

```

Notes on Example 7-13 on page 156:

- ▶ To set the symbol list, the API specifies that it must be wrapped using a SymbolList object.
- ▶ The sendErrorResponse() method is the same as Example 7-7 on page 149.

Note: Because the buy and sell HTML pages are so similar, they use a common template. Values that are specific to both are substituted in using symbols.

For more information about documents and templates, see *CICS Application Programming Guide*, SC34-6231.

7.3.8 Setting up TraderPL

In this section, we tell you how to install and run TraderPL.

Installing TraderPL

Similar to the source code and compiled classes for this example are in Trader-1.jar. Following the instructions in Chapter 3, “Setting up CICS to run Java applications” on page 37, make sure this jar file is added to the CICS Java classpath.

The new CICS resources that you must define alongside the existing definitions in the TRADER group are:

- ▶ PROGRAM(TRADERPJ)
CONCURRENCY(Threadsafe)
JVM(Yes)
JVMCLASS(com.ibm.itso.sg245275.trader.TraderPL)
- ▶ DOCTEMPLATE(TRADBYSL)
TEMPLATENAME(TRADBYSL)
DDNAME(DFHHTML)
MEMBERNAME(TRADBYSL)

- ▶ DOCTEMPLATE(TRADCOMD)
TEMPLATENAME(TRADCOMD)
DDNAME(DFHHTML)
MEMBERNAME(TRADCOMD)
- ▶ DOCTEMPLATE(TRADCOML)
TEMPLATENAME(TRADCOML)
DDNAME(DFHHTML)
MEMBERNAME(TRADCOML)
- ▶ DOCTEMPLATE(TRADLOGF)
TEMPLATENAME(TRADLOGF)
DDNAME(DFHHTML)
MEMBERNAME(TRADLOGF)

Make sure that the corresponding HTML templates exist in CICSSYSF.APPL.TEMPLATE.

Note: DFHHTML is the DD card for template data sets in the startup JCL. In our example, it is set to CICSSYSF.APPL.TEMPLATE.

To polish up the look of the Web pages, a CICS hosted image is added to the HTML that uses the ImageLoader program to load the file RBHOME.gif. Ensure that the relevant DOCTEMPLATE was installed and that the image exists in the template data set (it needs to be uploaded as a binary file). Notice that for images, the attributes APPENDCRLF(No) and TYPE(Binary) must be set in the DOCTEMPLATE definition:

- ▶ DOCTEMPLATE(RBHOME)
TEMPLATENAME(RBHOME)
DDNAME(DFHHTML)
MEMBERNAME(RBHOME)
APPENDCRLF(No)
TYPE(Binary)

Tip: You can use the ImageLoader program to host images in CICS by adding the following statement to your HTML:

```
<IMG src='imageldr?filename=<filename>&filetype=<filetype>' border='0'>
```

In the statement:

- ▶ *file name:* Case sensitive name of the DOCTEMPLATE definition for the image file.
- ▶ *file type:* Image type of the file, for example, gif, jpg, bmp.

Running TraderPL

Similar to “Running SimpleTraderPL” on page 150, bring up a Web browser and type in the URL:

```
http://wtsc66.itso.ibm.com:5275/cics/cwba/traderpj
```

Ensure that the port number matches up to that in your TCPIP SERVICE definition and that the program name at the end is traderpj, as shown in Figure 7-10 on page 159.

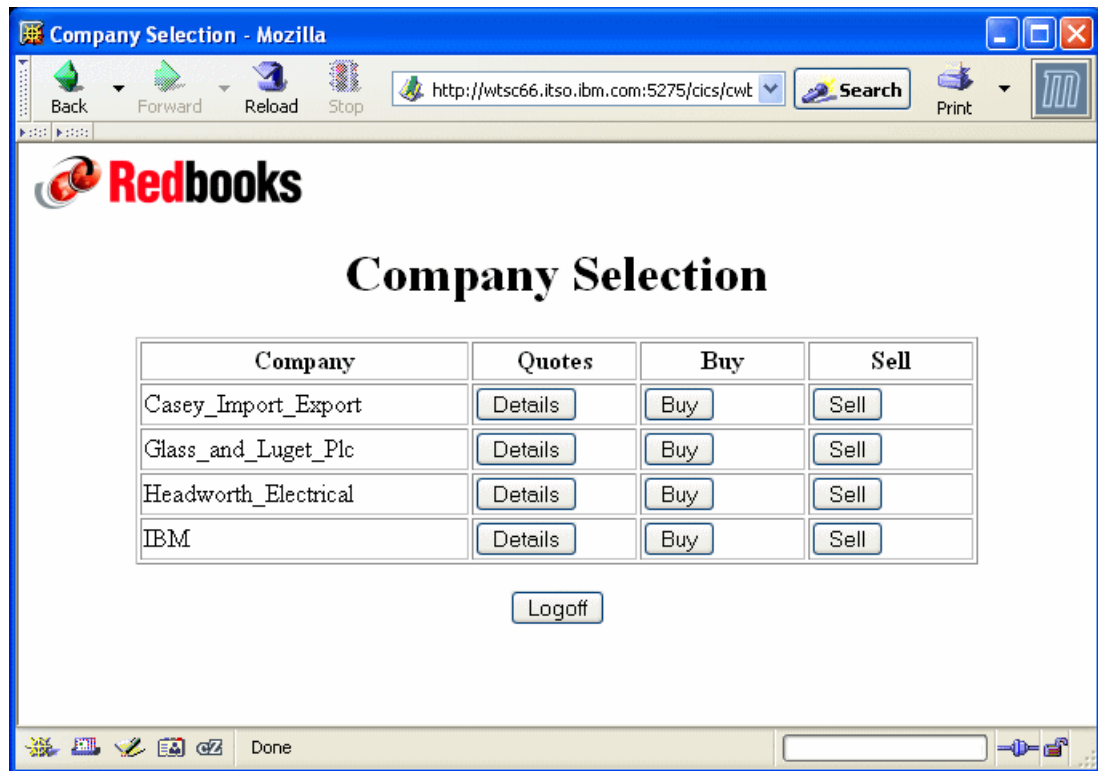


Figure 7-10 The company selection screen

If an error occurs, for example, if TRADERBL is not installed, then a message is displayed, as shown in Figure 7-11 on page 160.

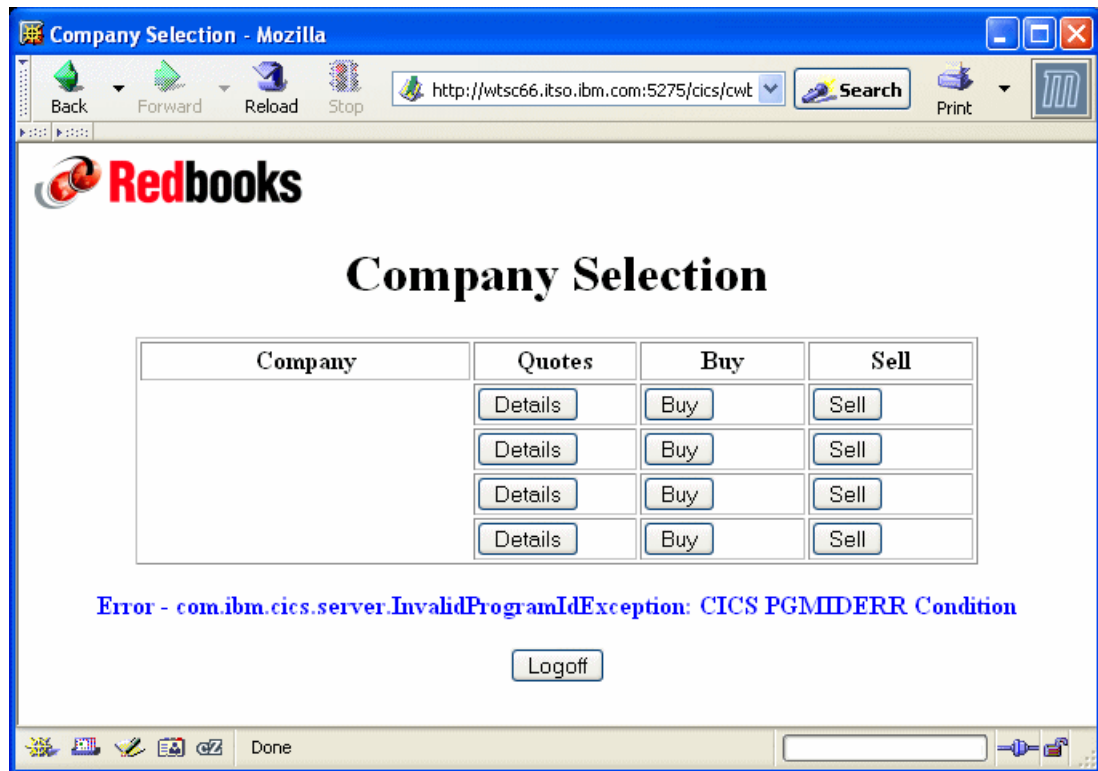


Figure 7-11 The message received if TRADERBL was not installed

Congratulations. You implemented a Web interface to your existing COBOL heritage application.

Tip: To prove that there are no smoke and mirrors involved, try to run the Web interface and the original 3270 display side-by-side, and watch the figures get updated in both.

7.3.9 Web security

Enabling security in the TraderPL program does not require any additional coding because CICS takes all of the pain out of it by handling it for us. All you must do is set the AUTHENTICATE attribute to Basic in the TCPIP SERVICE definition.

Note: For information about basic authentication and other forms of Web security, see *CICS Internet Guide*, SC34-6245.

Now when TRADERPJ is accessed over the Web, you are prompted with a login box. Ensure that you enter a valid user name and password combination, as shown in Figure 7-12 on page 161.

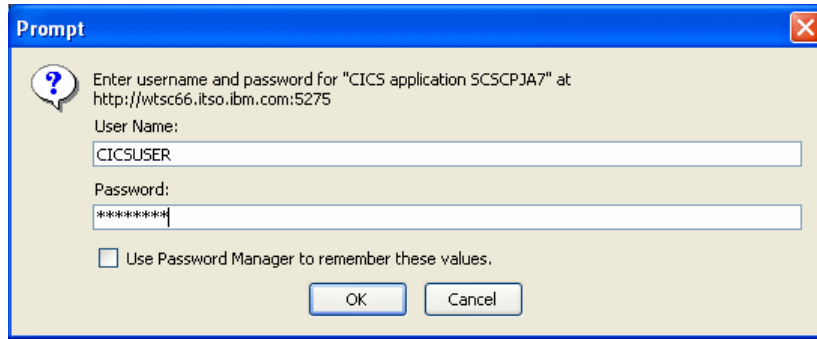


Figure 7-12 Log in to run TraderPL

The JCICS call `Task.getTask().getUserID()` picks up this user ID, and TRADERPJ continues seamlessly with security enabled.

Because there is no explicit way of closing an authenticated Web session in CICS, the browser window must be closed to perform a logoff.

Note: For enhanced levels of Web security, refer to *CICS RACF Security Guide*, SC34-6249.

7.4 Migrating TRADERBL to JCICS

A Web front-end introduces an application to a world-wide client base. Originally, the COBOL heritage application was probably written for an intended user base of no more than a few hundred people, which through the Web can become millions. Knowing this, it might be the case that the back-end code might not handle as well when scaled up to this factor. In anticipation of this, the approach is to re-implement the back-end using JCICS, and separate out the logic and data access functionality into two components. Then, if needed, the data access is migrated to a DB2 back end or even a workload balanced solution. Figure 7-13 on page 162 illustrates the migration of the back end to JCICS.

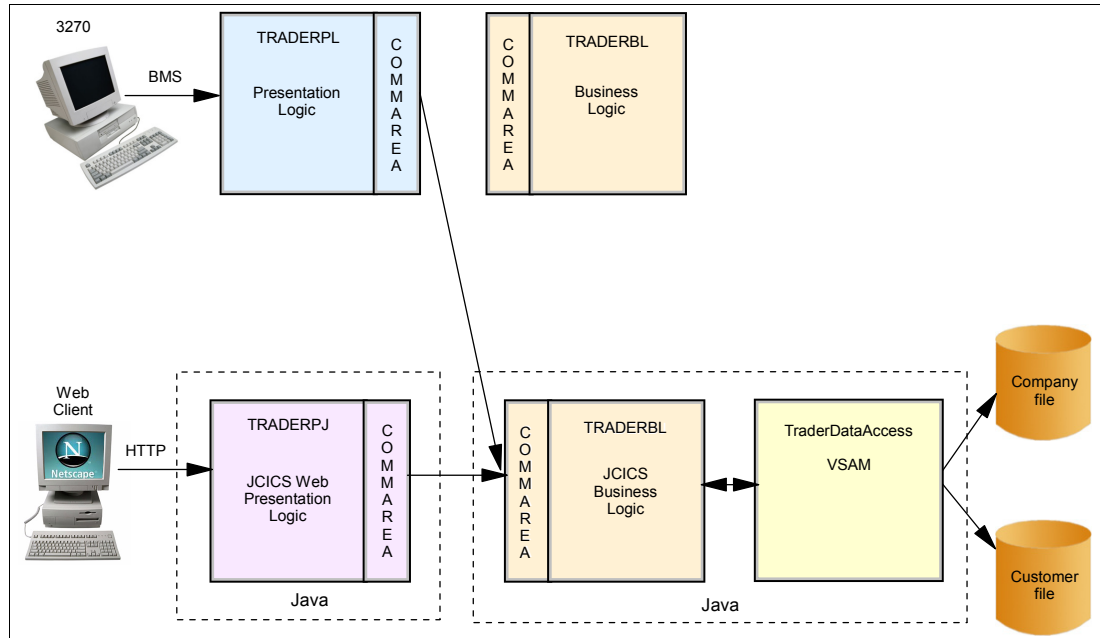


Figure 7-13 Migrating the back-end to JCICS

7.4.1 Mapping COBOL to Java

Instead of re-architecting the code in a pure object-oriented (OO) setup, the alternative is to re-code it with a procedural Java equivalent, which means to write Java in a procedural and not OO manner. In this way, if a problem occurs in the code, you can be debug it through comparison with the original working COBOL version.

Note: Converting COBOL to an object-oriented Java setup is possible, but it requires a complete understanding of the code, which might not be a trivial task because if there is a lot of code, there is also a lot to understand.

Fundamentally, COBOL was designed to be written in a human understandable form, so reading the code is just like reading an instruction manual. By migrating the back-end logic to Java, you get an understanding of how the languages can be mapped.

Working storage

The first section in TRADERBL is the working storage. You can categorize the various COBOL data structures in it as:

- ▶ Error messages
- ▶ Input/output buffers
- ▶ Constants
- ▶ Conversion fields
- ▶ Miscellaneous

Error messages

Error messages feed debug information to the user. Example 7-14 on page 163 shows the COMPANY-NOT-FOUND message. Conveniently these map directly to Java static constants, as shown in Example 7-15 on page 163.

Example 7-14 The company not found message

```
03 COMPANY-NOT-FOUND-MSG.  
   05 FILLER PIC X(25) VALUE 'COMPANY #CCCCCCCCCCCCCCCC'.  
   05 FILLER PIC X(25) VALUE 'CCC NOT FOUND'.
```

Input/output buffers

The following data areas are used to read and write data using CICS API:

- ▶ CUSTOMER-IO-BUFFER
- ▶ COMPANY-IO-BUFFER
- ▶ COMMAREA-BUFFER

Using the same approach as 7.3.1, “Wrapping the COMMAREA” on page 139, the classes CustomerIOWrapper and CompanyIOWrapper are written, and the class CommareaWrapper is reused from SimpleTraderPL. These become instance variables in TraderBL, as shown in Example 7-15.

Constants

Various constants are defined to represent return values for actions in the code. We used the same values in TraderPL, and we reuse them here to minimize code and to alleviate any potential errors when passing the values between TraderPL and TraderBL. We first mentioned the TraderConstants class in “SimpleTraderPL” on page 148.

Conversion fields

One of the powerful features of COBOL is its ability to convert between and format multiple data types. The conversion fields in TRADERBL, a sample of which is shown in Example 7-24 on page 169, show how you can redefine data areas to allow convenient conversions between character-based numeric types.

To replicate this behavior in Java, you can do one of these:

- ▶ Wrap the conversion fields using the approach in “Wrapping the COMMAREA” on page 139, and implement specific conversion and formatting rules in the various getters and setters.
- ▶ Utilize the powerful conversion and formatting facilities that are available to Java.

To reduce the number of wrapper classes, and fundamentally the amount of code, TraderBL uses the second option and inserts explicit Java code to perform the conversions. See Example 7-25 on page 169 for an example of this.

Miscellaneous

What remains are helper values that you can easily map to Java types and instantiate as local variables when needed. The tracing fields are not implemented because Java exception data provides enough information about problems.

Putting it together

Following a similar naming convention to TraderPL, the Java code for the back-end is in the class TraderBL. Example 7-15 shows the Java version of the working storage.

Example 7-15 TraderBL's equivalent to WORKING-STORAGE

```
private static final String COMPANY_NOT_FOUND_MSG =  
    "COMPANY #CCCCCCCCCCCCCCCC NOT FOUND"; // (1)  
  
private static final String REQUEST_NOT_FOUND_MSG =  
    "REQUEST CODE OF #RRRRRRRRRRRRRR INVALID";
```

```

private static final String SUB_FUNCTION_NOT_FOUND_MSG =
    "FUNCTION BUY/SELL CALLED WITH AN INVALID SUBCODE";

private static final String OVERFLOW_MSG =
    "OVERFLOW WHEN CALCULATING SHARE VALUE";

private static final String TOO_MANY_SHARES_MSG =
    "CUSTOMER TRIED TO SELL MORE SHARES THAN THEY OWN";

private static final String NO_SHARES_MSG =
    "CUSTOMER HAS NO SHARES TO SELL IN SELECTED COMPANY";

private static final String VALIDATE_MSG =
    "VALIDATING COMPANY #CCCCCCCCCCCCCCCC";

private static final String TOO_MANY_MSG =
    "TOO MANY SHARES REQUESTED, MAX OWNERSHIP IS 9999";

private final CommareaWrapper fCommareaWrapper = new CommareaWrapper();
private final CustomerIOWrapper fCustomerIOWrapper = new CustomerIOWrapper();
private final CompanyIOWrapper fCompanyIOWrapper = new CompanyIOWrapper();

```

Note on Example 7-15 on page 163:

- A further expansion is to externalize these strings and provide internationalized versions that can be returned depending on the language preference in the HTTP header.

Abstracting the data access

With the wrappers written, we have everything that we need to design the data access interface. There are five sections in the COBOL program that use file access for retrieving data. They are replaced with a call to an object that implements the Java interface `TraderDataAccess`. This interface, shown in Example 7-16, provides an individual method for each type of data access required in the program.

Because Java does not have an equivalent to EIBRESP, the methods return the type `int`, which contains the response value. If a class that implements the interface does not directly support these values, it maps them back to ensure compatibility with `TraderBL`. The response values that we use in `TraderBL` are included as static constants, and we use `DFHRESP_UNKNOWN` for values that are not listed.

We discuss an implementation of this interface for VSAM data access in 7.4.2, “Using `TraderBL` with VSAM” on page 169.

Example 7-16 The `TraderDataAccess` interface

```

public interface TraderDataAccess {
    public static final int DFHRESP_UNKNOWN = -1;
    public static final int DFHRESP_NORMAL = 0;
    public static final int DFHRESP_NOTFND = 13;

    public int getCompanyNames(CommareaWrapper wrapper, String key);
    public int readCustfile(CustomerIOWrapper wrapper, String key);
    public int readCustfileForUpdate(CustomerIOWrapper wrapper, String key);
    public int writeCustfile(CustomerIOWrapper wrapper, String key);
    public int readCompfile(CompanyIOWrapper wrapper, String key);
    public int rewriteCustfile(CustomerIOWrapper wrapper);           // (1)
}

```

Note on Example 7-16 on page 164:

- Does not need a key because the file is already opened from a previous `readCustfileForUpdate()` call.

Main section

The main section is the equivalent of `public static void main()` in a Java class, which gets executed when the program starts. Because `TraderBL` is written procedurally to reflect the COBOL code, this is the likely place to put the MAINLINE section. Observing the recommended approach 2.3.2, “Continuous JVM” on page 21, the code instead is placed in `TraderBL`’s constructor and the object instantiated from this method, which minimizes the need for static variables.

To allow for pluggable data access, a class that implements the `TraderDataAccess` interface is passed to the object constructor and the supplied COMMAREA data.

Example 7-17 shows the syntax to instantiate the `TraderBL` object in the main method.

Example 7-17 Instantiating the TraderBL object in the main method

```
public static void main(CommAreaHolder commAreaHolder) {  
    new TraderBL(commAreaHolder, new TraderVSAMAccess());           // (1)  
}
```

Note on Example 7-16 on page 164:

- For more information about `TraderVSAMAccess`, see “Using `TraderBL` with VSAM” on page 169.

Mainline section

In `TRADERBL`, the MAINLINE section evaluates the request that is passed in the COMMAREA and executes the relevant code section before exiting the program.

Writeq-TS

At the start of MAINLINE, and in many places throughout the code, messages are sent to a TSQ by moving text to a variable and then calling `WRITEQ-TS`. In the Java implementation, this is collapsed to a single call to `writeMessage()` and the message passed as a parameter. Similar to `WRITEQ-TS`, the text is formatted and pre-pended with a time stamp. Instead of having separate sections for doing this they are included in-line instead.

Note: Not all EXEC CICS calls have a JCICS equivalent. In some cases, such as EXEC CICS ASKTIME, you must implement a Java equivalent.

For convenience, the message is outputted to Java’s `stdout` instead of a TSQ, which allows an unlimited length log to be used. If you still want the output in a TSQ, this is possible using the fully supported temporary storage JCICS API.

Example 7-18 The writeMessage() method

```
private void writeMessage(String comment) {  
    // Get a time stamp  
    final String timeStamp = new SimpleDateFormat("h:mm:ss a").format(new Date());  
  
    // If they exist, replace fields in the comment with COMMAREA values  
    comment =  
        comment.replaceFirst("#CCCCCCCCCCCCCCCC", fCommareaWrapper.getCompanyName());
```

```

comment =
    comment.replaceFirst("#RRRRRRRRRRRRR", fCommareaWrapper.getRequestType());

comment =
    comment.replaceFirst("#UUUUUUUUUUUUUU",
        fCommareaWrapper.getUserId().substring(0, 15));

comment =
    comment.replaceFirst("#R", fCommareaWrapper.getReturnValue());

// Remove spaces from the comment
comment.trim();

// Original wrote to TSQ, for Java we'll use stdout
System.out.println(timeStamp + " TraderBL: " + comment);
}

```

Tip: Use `tail -f <filename>` to continually display Java stdout or stderr in a ssh or telnet terminal.

Evaluate statement

Evaluate statements in COBOL allow for multiple outcomes to be coded dependant on a variable's value, as shown in Example 7-19.

Example 7-19 A COBOL evaluate statement

```

EVALUATE REQUEST-TYPE
    WHEN GET-COMPANY-REQ
        PERFORM GET-COMPANY
    WHEN SHARE-VALUE-REQ
        PERFORM GET-SHARE-VALUE
END-EVALUATE.

```

The Java equivalent to evaluate statements are switch statements. The only difference is that switch statements are limited to integer-based or character-based comparisons and so cannot be used with string-based values. Instead, you can use a series of if-then-else statements, which we show in Example 7-20 on page 167.

Tip: Good practice for Java string comparisons is to call *equals* on the constant and not the variable (for example, `CONSTANT.equals(astring)`), which protects against null pointer exceptions if *astring* is set to null.

Perform <section>

Depending on the request type, different actions are performed in the evaluate statement, as shown in Example 7-19. The `PERFORM GET-COMPANY` call executes code that resides in the `GET-COMPANY` section, which in Java equates to calling a method with return type void. So any `PERFORM <section>` calls in the COBOL code are converted such that `PERFORM GET-COMPANY` becomes `getCompany()`.

Note: COBOL allows the hyphen character (-) in identifiers, while Java does not. The Java convention is to use inner capitalization, so `GET-COMPANY` in COBOL becomes `getCompany()` in Java.

Putting it together

Following the directions that we discussed so far, the MAINLINE code in TraderBL becomes the code in Example 7-20.

Example 7-20 The TraderBL constructor

```
private final TraderDataAccess fTraderDataAccess;                                // (1)

public TraderBL(CommAreaHolder commAreaHolder, TraderDataAccess traderDataAccess) {
    // Set the traderDataAccess reference
    fTraderDataAccess = traderDataAccess;

    writeComment("Entry");

    // Get the commarea byte[] and put in wrapper
    fCommareaWrapper.setByteArray(commAreaHolder.value);

    // Evaluate the request type
    final String requestType = fCommareaWrapper.getRequestType();

    if (TraderConstants.GET_COMPANY_REQ.equals(requestType))    getCompany();
    else if (TraderConstants.SHARE_VALUE_REQ.equals(requestType)) getShareValue();
    else if (TraderConstants.BUY_SELL_REQ.equals(requestType))  buySell();
    else {
        fCommareaWrapper.setReturnValue(TraderConstants.UNKNOWN_REQUEST);
        writeComment(REQUEST_NOT_FOUND_MSG);
    }

    writeComment("Exit");

    // EXEC CICS RETURN END-EXEC                                                // (2)
}
```

Notes on Example 7-20:

- ▶ This holds a reference to the pluggable data access component passed in during object construction.
- ▶ There is no JCICS equivalent to EXEC CICS RETURN. Java programs that run in CICS must always run to the end and terminate cleanly to allow for appropriate cleanup.

Get company

The GET-COMPANY section is the first section to include file access functionality. It also has a few other commands worth mentioning.

Move

To move a value into a variable is to set its value. In Java, this is the same as using the = operator. Pay close attention to the code because COBOL implicitly converts values according to the data type that they are inserted into. In Java, you must do this explicitly. Also, double check the value that you set because sometimes it is not obvious which wrapper the variable is in, and you might check the working-storage section for clarification.

Perform varying

Although not used in the Java version of getCompany() because file access is abstracted out to a TraderDataAccess implementation, we still want to mention the PERFORM VARYING command. This command is a flexible statement that is essentially the super set of the Java iteration statements, that is, both *for* and *while* loops can be represented with the PERFORM

statement. The main identifier, as to whether a *for loop* or a *do while* loop are used, is the WITH TEST BEFORE and WITH TEST AFTER modifiers, respectively. The default action is to use WITH TEST BEFORE.

The Java version

With the use of the pluggable data access component, the Java `getCompany()` method becomes substantially smaller than the COBOL version.

Example 7-21 The `getCompany()` method

```
private void getCompany() {  
    fCommareaWrapper.setCompanyName("");  
  
    fTraderDataAccess.getCompanyNames(fCommareaWrapper);  
}
```

Calculate shares bought

The CALCULATE-SHARES-BOUGHT section, shown in Example 7-22, takes the number of shares that are bought and adds it to the amount of shares owned. Both of these numbers are character-based numerics. By looking at a specific character in the total value, it is determined whether there is an overflow.

Example 7-22 The `calculate-shares-bought` section

```
CALCULATE-SHARES-BOUGHT SECTION.  
  ADD NO-OF-SHARES-DEC TO DEC-NO-SHARES GIVING SHARES-WORK1  
  EVALUATE SHARES-OVERFLOW  
    WHEN 0  
      MOVE SHARES-NORMAL TO NO-OF-SHARES-DEC  
      MOVE SHARES-NORMAL TO DEC-NO-SHARES  
      PERFORM UPDATE-BUY-SELL-FIELDS  
    WHEN OTHER  
      MOVE INVALID-BUY TO RETURN-VALUE  
      MOVE TOO-MANY-MSG TO COMMENT-FIELD  
      PERFORM WRITEQ-TS  
  END-EVALUATE  
.  
CALCULATE-SHARES-BOUGHT-EXIT.  
EXIT.
```

As we mentioned in “Conversion fields” on page 163, COBOL can convert and format values implicitly based on their definitions. Building on this, it also allows basic arithmetic to be performed on values that differ in type, which means that character-based numbers can be added or subtracted to binary-based numbers without the user implementing additional conversion code.

In Java, the share values are stored internally as strings, so numeric addition is not directly possible. Instead, the numbers are converted to integers, using the `Integer.parseInt()` method and then added together. To determine if the value overflowed, the integer total is evaluated using an if statement, which gives the `calculateSharesBought()` method shown in Example 7-23.

Example 7-23 The `calculateSharesBought()` method

```
private void calculateSharesBought() {  
    int sharesWork =  
        Integer.parseInt(fCommareaWrapper.getNoOfSharesDec()) +  
        Integer.parseInt(fCustomerIOWrapper.getDecNoShares());  
}
```

```

    if (sharesWork <= 9999) {
        fCommareaWrapper.setNoOfSharesDec("" + sharesWork);           // (1)
        fCustomerIOWrapper.setDecNoShares("" + sharesWork);
        updateBuySellFields();
    }
    else {
        fCommareaWrapper.setReturnValue(TraderConstants.INVALID_SALE_OR_BUY);
        writeMessage(TOO_MANY_MSG);
    }
}

```

Note on Example 7-23 on page 168:

- To insert the values into the wrappers they are converted to strings first. The Java compiler implicitly converts `"" + sharesWork` to `new String(sharesWork)`.

Calculating share value

Calculating share value is when values are moved between different data types to perform arithmetic operations and number formatting. The code in Example 7-24 allows a number to be inserted into NUM-VALUE, which replaces any leading spaces with the 0 character. When CHAR-VALUE is referenced, it picks up this number formatting.

Example 7-24 COBOL values used for conversions

```

03 CONVERSION-FIELDS.
   05 CHAR-VALUE.
       07 CHAR-INT-PART      PIC X(09).
       07 FILLER             PIC X VALUE ' '.
       07 CHAR-DEC-PART     PIC X(02).
   05 NUM-VALUE REDEFINES CHAR-VALUE.
       07 NUM-INT-PART      PIC 9(09).
       07 GUB92             PIC X.
       07 NUM-DEC-PART     PIC 9(02).

```

Because arithmetic in Java occurs using binary-based numbers, the `DecimalFormat` class is used to format it similar to the COBOL code, as shown in Example 7-25. The result is placed in the `COMMAREA` wrapper.

Example 7-25 Formatting a number as a string

```

final DecimalFormat decimalFormat = new DecimalFormat("000000000.00");
fCommareaWrapper.setTotalShareValue(decimalFormat.format(totalShareValue));

```

Housekeeping

It is most likely that `TRADERBL` is an application that evolved over many years. Through writing its JCICS equivalent, notice that various statements were commented out, and sections, such as `BUY-SELL-UPDATE`, are no longer used by the presentation layer. This allows for a little spring cleaning, and so redundant code is removed along the way.

7.4.2 Using TraderBL with VSAM

The result of 7.4.1, “Mapping COBOL to Java” on page 162 gives us a JCICS version of `TRADERBL` that has a pluggable interface to access the client and company data. To access the VSAM data, the class `TraderVSAMAccess` is used, which implements the `TraderDataAccess` interface, thus giving direct access to the existing VSAM data using the JCICS file access API.

The VSAM data access component

Example 7-16 on page 164 shows the methods to implement. Each method takes, as a parameter, a wrapper data area within which to return the results and a key for accessing a specific record. Similar to the REWRITE-CUSTFILE COBOL code, `rewriteCustfile()` does not take a key because the file was already opened using a `readCustfileForUpdate()` call. Example 7-26 shows CICS file access code from the READ-COMPFIL section in TRADERBL.

Example 7-26 File access code in the read-compfile section

```
EXEC CICS READ
  FILE('COMPFILE')
  INTO(COMPANY-IO-BUFFER)
  LENGTH(LENGTH OF COMPANY-IO-BUFFER)
  RIDFLD(COMPANY-NAME OF COMMAREA-BUFFER)
  NOHANDLE
END-EXEC
```

Because `TraderVSAMAccess` is explicitly written to access the files `CUSTFILE` and `COMPFILE`, the file names are stored inside the class as static constants. The appropriate value is then set on the `JCICS KSDS` object to access the data, as shown in Example 7-27.

Example 7-27 The `readCompfile()` method

```
public int readCompfile(CompanyIOWrapper wrapper, String key) {
    int result = DFHRESP_NORMAL;

    final RecordHolder recordHolder = new RecordHolder();
    final KSDS klds = new KSDS();

    klds.setName(COMPFILE);

    try {
        klds.read(key.getBytes(), recordHolder);
        wrapper.setByteArray(recordHolder.value);
    }
    catch (RecordNotFoundException e) { result = DFHRESP_NOTFND; }
    catch (Exception e) {
        result = DFHRESP_UNKNOWN;
        System.err.println("Exception: " + e);
        e.printStackTrace();
    }

    return result;
}
```

All of the methods in `TraderVSAMAccess` are implemented similar to this using the `JCICS` file API. Observe that, as specified in the design, no business logic is performed in this class; instead, it is solely used to provide access to the VSAM data files.

7.4.3 Setting up TraderBL

In this section, we tell you how to install and run `TraderPL` with `TraderBL` as the back-end.

Installing TraderPL with TraderBL as the back end

The source code and compiled classes for this example are in Trader-2.jar. To install TraderPL with TraderBL as the back end:

1. Following the instructions in Chapter 3, “Setting up CICS to run Java applications” on page 37, and make sure that this jar file replaces Trader-1.jar in the CICS Java classpath.
2. In CICS, run the command CEMT SET JVMPOOL PHASE to phase out any JVMs that still might reference the old code.
3. Re-define the existing TRADERBL CICS resource definition such that it has the following properties:
 - PROGRAM(TRADERBL)
 - CONCURRENCY(Threadsafe)
 - JVM(Yes)
 - JVMCLASS(com.ibm.itso.sg245275.trader.TraderBL)
4. Run CEMT SET PROGRAM(TRADERBL) NEWCOPY to pick up the new program definition.

Note: Before you deploy your newly developed back end on a live system, thoroughly test it against sample or replica data.

Running TraderBL

Because there is no change to the front-end code, open a Web browser, and type in the URL:

`http://wtsc66.itso.ibm.com:5275/cics/cwba/traderpj`

Ensure that the port number matches that specified in your TCPIP SERVICE definition and that the program name at the end is traderpj.

You will see the same results, as shown in Figure 7-10 on page 159. This time, however, the back-end code being driven is that of the Java class TraderBL. To prove this, look at Java's stdout to observe the various messages output during the run. Example 7-28 shows the message's output for a `getCompany()` call.

Example 7-28 Messages for a `getCompany()` call

```
12:07:49 PM TraderBL: Entry
12:07:49 PM TraderBL: Entry to GET-SHARE-VALUE
12:07:49 PM TraderBL: Reading record from customer file
12:07:49 PM TraderBL: Reading record from company file
12:07:49 PM TraderBL: Building return commarea
12:07:49 PM TraderBL: Exit
```

Now try the same thing with the original 3270 front end. Nothing appears to have changed, but by watching the Java stdout messages you can see that the TraderBL implementation of the business logic is driven under the covers instead.

Congratulations. You migrated your COBOL heritage application to a JCICS environment with a scalable back end.

7.5 Moving to a DB2 back end

With the Web interface and JCICS back end in place, the next stage is to replace the existing VSAM data setup with a DB2 implementation. The pluggable design that we implemented in

7.4, “Migrating TRADERBL to JCICS” on page 161, combined with an understanding of DB2 JDBC™, aids this in being a trivial task. Figure 7-14 illustrates the implementation of a DB2 back end.

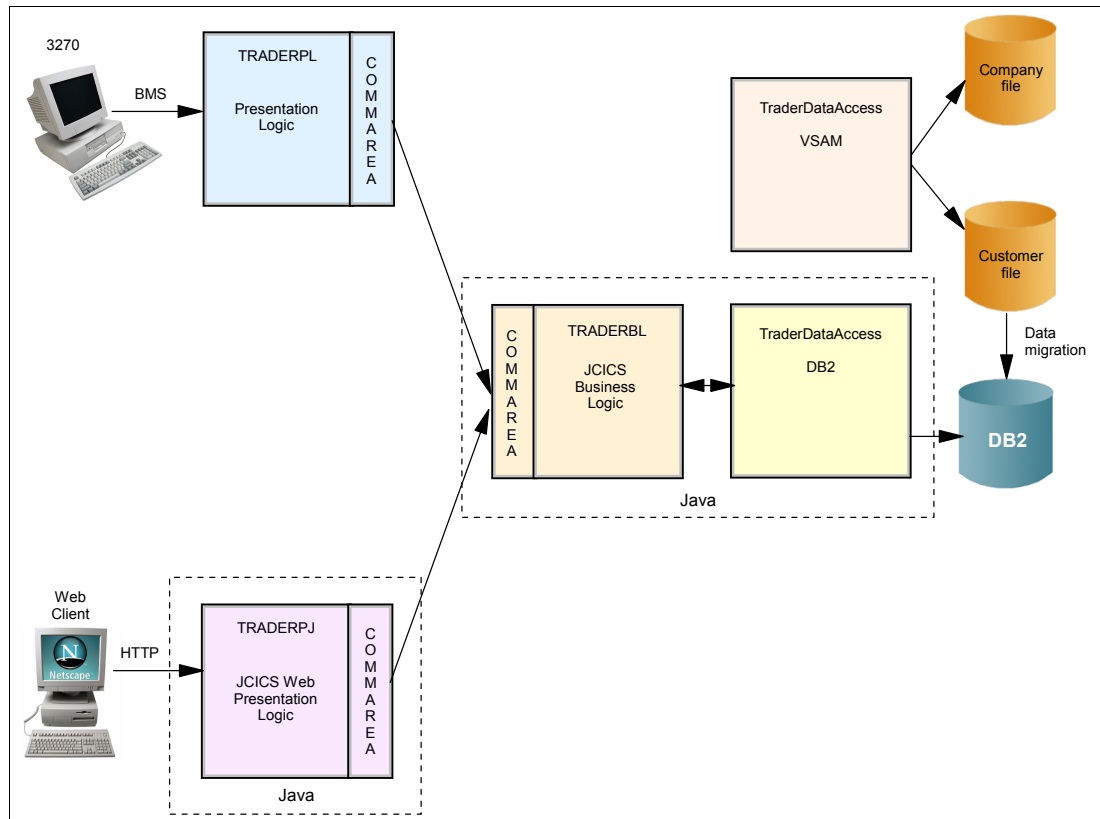


Figure 7-14 Implementing a DB2 back-end

7.5.1 Data migration

The client and company data in the VSAM files are stored as flat data; however, databases generally store elements in structured columns.

Database tables

The data in the VSAM file COMPDATA is broken down into the columns shown in Table 7-1.

Table 7-1 The TRADER_COMPANY table

Column name	Column type	Length	Nulls
name	Character	20	No
unitprice	Double	8	No
price1dayago	Double	8	No
price2daysago	Double	8	No
price3daysago	Double	8	No
price4daysago	Double	8	No
price5daysago	Double	8	No

Column name	Column type	Length	Nulls
price6daysago	Double	8	No
price7daysago	Double	8	No
sellcommission	Double	8	No
buycommission	Double	8	No

Similarly, the data in the VSAM file CUSTDATA is stored using the columns in Table 7-2.

Table 7-2 The TRADER_CUSTOMER table

Column name	Column type	Length	Nulls
username	Character	8	No
companyname	Character	20	No
sharesheld	Decimal	19,0	No

The TRADER_CUSTOMER table contains the minimum fields that are required to perform share transactions.

Data migration SQL

To simplify the data migration in this example, the company information is manually inserted into the appropriate table using the SQL statements in Example 7-29. The client information is not migrated, and so all share holdings are reset to zero. For a real-life migration, one might consider the *CICS VSAM Transparency* tool for transferring the data. See the following Web site for more information about this tool.

<http://www.ibm.com/software/hp/cics/vt/>

Example 7-29 SQL statements for inserting the data into TRADER_COMPANY

```

INSERT INTO TRADER_COMPANY VALUES('Casey_Import_Export',
                                   79, 77, 78, 72, 70, 65, 63, 59, 100, 0);
INSERT INTO TRADER_COMPANY VALUES('Glass_and_Luget_Plc',
                                   19, 22, 25, 20, 16, 20, 22, 17, 200, 0);
INSERT INTO TRADER_COMPANY VALUES('Headworth_Electrical',
                                   124, 131, 133, 133, 137, 138, 141, 300, 0);
INSERT INTO TRADER_COMPANY VALUES('IBM',
                                   163, 163, 162, 160, 161, 159, 156, 157, 400, 0);

```

7.5.2 Changing the JVM profile for DB2

To change the JVM profile for DB2:

1. Add the DB2 library into our LIBPATH_SUFFIX in JVM profile.
2. Add the DB2 JDBC driver classes into our CLASSPATH_SUFFIX in JVM profile.

Example 7-30 shows the JVM profile for DB2V9

Example 7-30 JVM profile for DB2V9

```

LIBPATH_SUFFIX=/usr/lpp/db2910/lib
CLASSPATH_SUFFIX= /usr/lpp/db2910/classes/db2jcc.jar:\
                  /usr/lpp/db2910/classes/db2jcc_javax.jar:\

```

For detailed information about what is required to support Java programs in the CICS DB2 environment, refer to *CICS Transaction Server for z/OS DB2 Guide*, SC34-6837.

Important: In this example, we use DB2V9, so we needed to migrate from using the JDBC/SQLJ Driver for OS/390 and z/OS to the IBM Data Server Driver for JDBC and SQLJ. If you are using the JDBC/SQLJ Driver for OS/390 and z/OS, read Chapter 9 in *DB2 Application programming Guide and Reference for Java*, SC18-9842.

7.5.3 Using TraderBL with DB2

With the pluggable architecture that we discussed in “Abstracting the data access” on page 164, to move TraderBL to a DB2 back end, we must create a Java class that implements the TraderDataAccess interface. The class, called TraderDB2Access, provides implementations of the methods shown in Example 7-16 on page 164, which performs DB2 calls for accessing the data.

Opening and closing the database connection

A common function that the methods in TraderDB2Access share is opening and closing a DB2 connection. The code for this is put into two utility methods, as shown in Example 7-31.

Example 7-31 The openConnection() and closeConnection() methods

```
private Connection fConnection = null;                                // (1)

private int openConnection() {
    int result = DFHRESP_NORMAL;

    try {
        final String jdbcUrl = "jdbc:default:connection";           // (2)

        fConnection = DriverManager.getConnection(jdbcUrl);
        fConnection.setAutoCommit(false);
    }
    catch (Exception e) {
        result = DFHRESP_UNKNOWN;
        System.err.print("Exception: " + e);
        e.printStackTrace();
    }

    return result;
}

private int closeConnection() {
    int result = DFHRESP_NORMAL;

    try {
        fConnection.close();
        fConnection = null;
    }
    catch (Exception e) {
        result = DFHRESP_UNKNOWN;
        System.err.print("Exception: " + e);
        e.printStackTrace();
    }
}
```



```
    return result;  
}
```

Notes on Example 7-31 on page 174:

- ▶ Holds a reference to the connection object, which is used for database access.
- ▶ The default JDBC connection is used.
- ▶ We have not used a DataSource for accessing DB2. If you want to know about DataSources and how they are used to connect to DB2, refer to the manual: DB2 V9 Application Programming Guide and Reference for Java.

Note: We used JDBC for the DB2 access here, but you must really consider using SQLJ programs as the preferred method for accessing DB2 from CICS. SQLJ programs must be bound into plans, and it is here that the SQL validation takes place compared to JDBC where the SQL is dynamically validated and bound at execution time. If you require the flexibility, then you must use JDBC; otherwise, SQLJ must be your first choice. More information can be found in the CICS DB2 Guide and DB2 Application Programming Guide and Reference for Java.

The SQL statements

SQL statements are needed to extract and update information in the database tables, as shown in Example 7-32. For convenience, the SQL is stored as static variables in TraderDB2Access. Use the prepared statement functionality to update the values dynamically.

Example 7-32 SQL to extract and update the data

```
private static final String SQL_GET_COMPANY_NAMES =  
    "SELECT name FROM trader_company";  
  
private static final String SQL_GET_COMPANY_DATA =  
    "SELECT * FROM trader_company WHERE name=?";           // (1)  
  
private static final String SQL_GET_CUSTOMER_DATA =  
    "SELECT sharesheld FROM trader_customer WHERE username=? AND companyname=?";  
  
private static final String SQL_UPDATE_CUSTOMER_DATA =  
    "UPDATE trader_customer SET sharesheld=? WHERE username=? AND companyname=?";  
  
private static final String SQL_INSERT_CUSTOMER_DATA =  
    "INSERT INTO trader_customer(username, companyname, sharesheld) VALUES(?,?,?)";
```

Note on Example 7-32:

- ▶ Values get inserted into the place of the question mark symbol (?).

Running a query

All the methods in TraderDB2Access follow the similar process pattern:

1. Open a database connection.
2. Prepare the SQL statement.
3. Run the query.
4. Extract the results, and insert them into supplied data wrapper, if required.
5. Close the database connection.
6. Return the response code.

One of the methods, `readCompfile()`, which implements this structure, is shown in Example 7-33.

Note that when you insert values into the `CompanyIOWrapper` in the `readCompfile()` method, you must format them first because the original `COMPANY-IO-BUFFER` did not enforce formatting; instead, the format was picked up from the way that the data was stored in the VSAM files. This means that you must format the data up front using a `DecimalFormat` object to provide data consistency. If `COMPANY-IO-BUFFER` specified the format of these values, it does not need to be done here; instead, they are automatically converted upon insertion.

Example 7-33 The `readCompfile()` method

```
public int readCompfile(CompanyIOWrapper wrapper, String key) {
    int result = openConnection();

    // If everything's ok then get the data
    if (result == DFHRESP_NORMAL) {
        PreparedStatement prepStmt = null;
        ResultSet resultSet = null;

        try {
            prepStmt = fConnection.prepareStatement(SQL_GET_COMPANY_DATA);
            prepStmt.setString(1, key);
            resultSet = prepStmt.executeQuery();                // (1)

            if (resultSet.next()) {
                final DecimalFormat df1 = new DecimalFormat("00000.00");
                final DecimalFormat df2 = new DecimalFormat("000");

                wrapper.setShareValue(df1.format(resultSet.getDouble("unitprice")));
                wrapper.setValue1(df1.format(resultSet.getDouble("price1daysago")));
                wrapper.setValue2(df1.format(resultSet.getDouble("price2daysago")));
                wrapper.setValue3(df1.format(resultSet.getDouble("price3daysago")));
                wrapper.setValue4(df1.format(resultSet.getDouble("price4daysago")));
                wrapper.setValue5(df1.format(resultSet.getDouble("price5daysago")));
                wrapper.setValue6(df1.format(resultSet.getDouble("price6daysago")));
                wrapper.setValue7(df1.format(resultSet.getDouble("price7daysago")));
                wrapper.setCommissionSell(df2.format(resultSet.getDouble("sellcommission")));
                wrapper.setCommissionBuy(df2.format(resultSet.getDouble("buycommission")));
            }
            else result = DFHRESP_NOTFND;                        // (2)
        }
        catch (SQLException e) {
            result = DFHRESP_NOTFND;
            System.err.print("Exception: " + e);
            e.printStackTrace();
        }
        finally {
            try {
                resultSet.close();
                prepStmt.close();
            } catch (Exception e) {}
        }
    }

    closeConnection();

    return result;
}
```

Notes on Example 7-33 on page 176:

- ▶ To retrieve information, we use the `executeQuery()`. To update data, you must use the `executeUpdate()` method.
- ▶ If no results were found, then set the response value accordingly.

Plugging in TraderDB2Access

The final step in the migration is to replace the code in `TraderBL` that instantiates a `TraderVSAMAccess` object with that of a `TraderDB2Access` object. The swap works seamlessly because both classes implement the `TraderDataAccess` interface. Example 7-34 shows the new `main()` method in `TraderBL`.

Example 7-34 The new `main()` method in `TraderBL`

```
public static void main(CommAreaHolder commAreaHolder) {  
    new TraderBL(commAreaHolder, new TraderDB2Access());  
}
```

You are now ready to try out the DB2 back end.

7.5.4 Setting up TraderBL with DB2

In this section, we tell you how to install and run `TraderPL` with `TraderBL` using DB2 for data access.

Installing TraderBL with DB2

The source code and compiled classes for this example are in `Trader-3.jar`. To install `TraderBL` with DB2:

1. Following the instructions in Chapter 3, “Setting up CICS to run Java applications” on page 37, make sure that this jar file replaces `Trader-2.jar` in the CICS Java classpath.
2. In CICS, run the command `CEMT SET JVMPOOL PHASE` to phase out any JVMs that still might reference the old code.
3. Do not forget to set up DB2, as discussed in 7.5.1, “Data migration” on page 172.

The new CICS resources that must be defined alongside the existing definitions in the `TRADER` group are:

- `DB2CONNECTOIN(TRADER) DB2ID(D9E1)`
- `DB2ENTRY(TRADER)`
`PLAN(DSNJDBC)`
- `DB2TRAN(TRADCOBL)`
`ENTRY(TRADER)`
`TRANSID(TRAD)`
- `DB2TRAN(TRADJAVA)`
`ENTRY(TRADER)`
`TRANSID(CWBA)`

4. Run `CEMT SET PROGRAM(TRADERBL) NEWCOPY` to pick up the new program definition that uses the `TraderDB2Access` object.

Attention: Make sure that the status of `DB2CONNECTION` is connected after you install `DB2CONNECTION`. Use the `CEMT INQUIRE DB2CONNECTION` command to do this.

Running TraderBL

To run TraderBL:

1. Because there is no change to the front-end code, open a Web browser, and type in the URL:
2. Ensure that the port number matches the port number that is specified in your TCPIP SERVICE definition and that the program name at the end is traderpj. You will see the same results as shown in Figure 7-10 on page 159.
3. Use the Web interface to buy some shares, and open up your favorite DB2 query tool.
4. Input and run the following SQL:

```
SELECT * FROM trader_customer;
```

The resulting data shows the shares that you just bought. Example 7-35 shows sample output.

Example 7-35 Query results

USERNAME	COMPANYNAME	SHARESHELD
CICSUSER	Glass_and_Luget_Plc	42.
CICSUSER	Casey_Import_Export	100.

5. Do the same thing with the original 3270 display to see that the back-end is fully implemented using DB2.

Congratulations. You migrated your JCICS Trader application to a DB2 back-end.

7.6 Adding a Web services interface

CICS Web services was introduced in CICS TS Version 3.1 and further enhanced in CICS TS Version 3.2. CICS SupportPac CA1P contains samples, tutorials, and step-by-step instructions about the function. Now, with the Web interface and JCICS back-end in place, the next stage is to provide a CICS Web services function to the Trader application.

7.6.1 Building the TraderBL Web service provider

The CICS Web services assistant is a CICS TS tool that generates the necessary Web service artifacts. The following exercise gives step-by-step instructions on how to configure CICS as a Web service requester using the CICS supplied tooling. IBM Rational Developer for System z product (RDz) provides an alternative tool. We do not describe that tool in this chapter.

To expose an existing CICS program as a Web service:

1. Run the CICS Web services assistant to generate the required files:
 - a. Create the zFS directories.
 - b. Extract the existing program interface.
 - c. Run the program DFHLS2WS.
 - d. Customize the service location in the WSDL.
2. Set up the CICS infrastructure:
 - a. Create a PIPELINE resource definition.

- b. Create a TCPIP SERVICE resource definition.
3. Deploy the Web service provider:
 - a. Deploy the Web service binding file.
 - b. Publish the WSDL to clients.
 - c. Test the Web service in Rational Developer for System z (RDz).

Running the CICS Web services assistant to generate the required files

CICS Transaction Server Version 3 provides the Web services assistant to generate Web services definitions (WSDL) from supplied program language structures and generate language structures from supplied WSDL documents. This assistant is actually two utility programs that run in z/OS batch. Both programs use the IBM Java SDK.

Creating the zFS directories

To expose a CICS program as a Web service over HTTP, a TCPIP SERVICE and a PIPELINE resource that is configured with a SOAP message handler are required. The PIPELINE resource definition points to zFS directories and also points to a pipeline configuration file.

More than one WEBSERVICE can share a single PIPELINE. We must, however, define a second PIPELINE for outbound requests because the pipeline configuration file referenced by a PIPELINE resource cannot be configured for both provider (inbound) and requester (outbound) services.

The set up requires configuration of files and directories on the zFS. The following directories are created:

- ▶ /u/cicsrs6/pipelines
- ▶ /u/cicsrs6/trader/shelf
- ▶ /u/cicsrs6/trader/wsbind/provider
- ▶ /u/cicsrs6/trader/wsbind/requester
- ▶ /u/cicsrs6/trader/wsd

Extracting the existing program interface

We are generating a new WSDL file from an existing program and using the DFHLS2WS utility (Language Structure to Web Service). As input, the DFHLS2WS utility must import the language copybooks that match the program's COMMAREA for request and response. The DFHLSWS utility can call a program using the new CICS Transaction Server Version 3 CHANNEL interface to support greater than 32 KB messages, but we are not using that facility.

The interface to program TraderBL is a COMMAREA. However, the DFHLS2WS utility does not support the REDEFINES and INDEX statements; therefore, to use DFHLS2WS, we must create a new copybook that has only those elements that relate to the program function. Example 7-36 shows the structure.

Example 7-36 The COMMAREA to TraderBL

```

01 COMMAREA-BUFFER.
   03 REQUEST-TYPE          PIC X(15) .
   03 RETURN-VALUE          PIC X(02) .
   03 USERID                PIC X(60) .
   03 USER-PASSWORD          PIC X(10) .
   03 COMPANY-NAME           PIC X(20) .
   03 CORRELID               PIC X(32) .
   03 UNIT-SHARE-VALUES.
```

```

05 UNIT-SHARE-PRICE      PIC X(08).
05 UNIT-VALUE-7-DAYS     PIC X(08).
05 UNIT-VALUE-6-DAYS     PIC X(08).
05 UNIT-VALUE-5-DAYS     PIC X(08).
05 UNIT-VALUE-4-DAYS     PIC X(08).
05 UNIT-VALUE-3-DAYS     PIC X(08).
05 UNIT-VALUE-2-DAYS     PIC X(08).
05 UNIT-VALUE-1-DAYS     PIC X(08).
03 COMMISSION-COST-SELL  PIC X(03).
03 COMMISSION-COST-BUY   PIC X(03).
03 SHARES.
    05 NO-OF-SHARES       PIC X(04).
03 TOTAL-SHARE-VALUE     PIC X(12).
03 BUY-SELL1             PIC X(04).
03 BUY-SELL-PRICE1       PIC X(08).
03 BUY-SELL2             PIC X(04).
03 BUY-SELL-PRICE2       PIC X(08).
03 BUY-SELL3             PIC X(04).
03 BUY-SELL-PRICE3       PIC X(08).
03 BUY-SELL4             PIC X(04).
03 BUY-SELL-PRICE4       PIC X(08).
03 ALARM-CHANGE          PIC X(03).
03 UPDATE-BUY-SELL       PIC X(01).
03 FILLER                PIC X(15).
03 COMPANY-NAME-BUFFER.
    05 COMPANY-NAME-TAB OCCURS 4 TIMES PIC X(20).

```

Running the program DFHLS2WS

The DFHLS2WS utility takes program language structures and generates WSbind and WSDL files, as shown in Figure 7-15 on page 181.

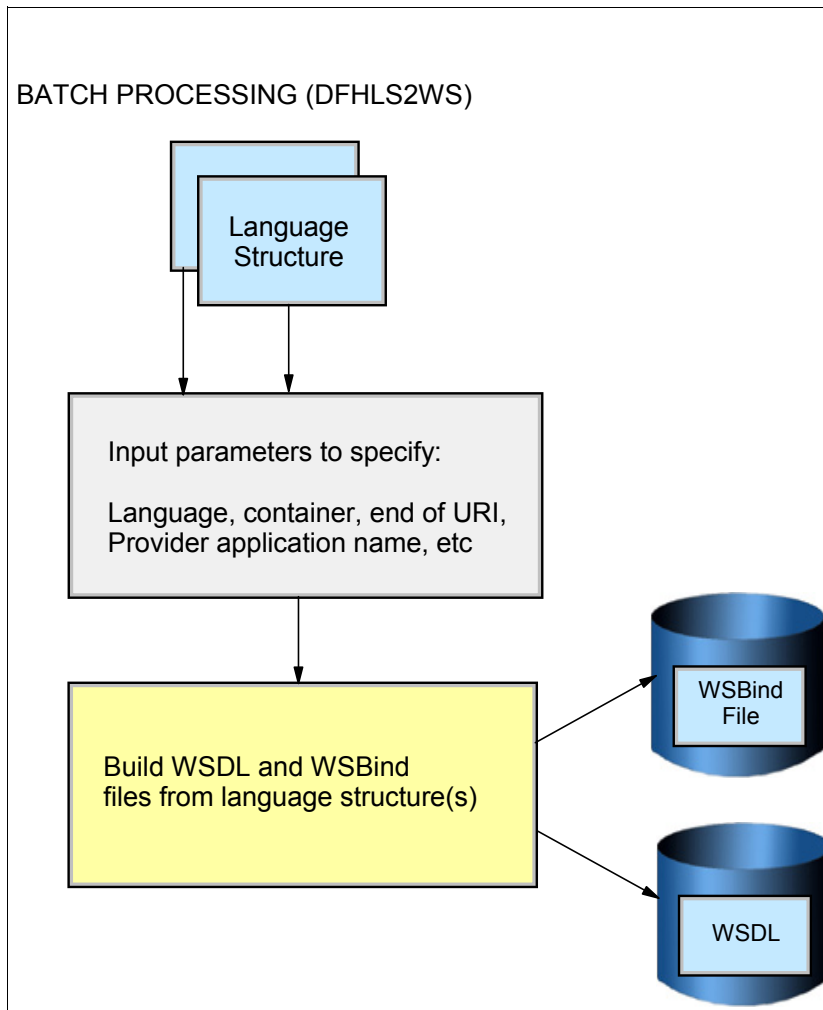


Figure 7-15 DFHLS2WS processing

The CICS supplied sample, JCL, to run DFHLS2WS is available on the zFS in the file /usr/lpp/cicsts/cicsts32/samples/webservices/JCL/LS2WS. We customized the JCL, as shown in Example 7-37.

Example 7-37 The DFHLS2WS JCL

```

//LS2WS      EXEC DFHLS2WS,
// TMPFILE='ls2ws',
// JAVADIR='/usr/lpp/java/J5.0',
// PATHPREF=' ',
// USSDIR='cicsts32'
//INPUT.SYSUT1 DD *
LOGFILE=/u/cicrs6/trader/wsbind/dfhls2ws.log
PDSLIB=//CICRS1.TRADER.COPYLIB
REQMEM=TRADERCM
RESPMEM=TRADERCM
LANG=COBOL
PGMNAME=TRADERBL
MAPPING-LEVEL=2.1
#TRANSACTION=CPIH
#CCSID=037
  
```

```
URI=trader/traderbl
PGMINT=COMMAREA
WSBIND=/u/cicsrs6/trader/wsbinding/provider/traderbl.wsbinding
WSDL=/u/cicsrs6/trader/wsd1/traderbl.wsd1
*/
```

We create the WSBinding file and WSDL file for Trader Web service. Based on the input parameters:

- ▶ The WSBinding file is placed in /u/cicsrs6/trader/wsbinding/provider and is called traderbl.wsbinding.
- ▶ The WSDL file is placed in /u/cicsrs6/trader/wsd1 and is called traderbl.wsd1.

Customizing the service location in the WSDL

As generated, the <service> element in the WSDL file is as shown in Example 7-38.

Example 7-38 Traderbl WSDL

```
<service name="TRADERBLService">
  <port binding="tns:TRADERBLHTTPSoapBinding" name="TRADERBLPort">
    <soap:address location="http://my-server:my-port/trader/traderbl"/>
  </port>
</service>
```

Make the following changes, which are needed before the Web service description can be published to clients:

- ▶ Replace my-service with wtsc66.itso.ibm.com
- ▶ Replace my-port with 5276

Setting up the CICS infrastructure

A PIPELINE resource definition contains a reference to a shelf directory, a WSDir directory containing WSBinding files, and a pipeline configuration file. If a value is provided for the WSDir directory, then CICS searches that directory, looking for WSBinding files, and dynamically installs WEBSERVICE and URIMAP resources. If no value is specified, you must manually define these resources. The pipeline configuration file is in XML format and describes the message handlers that act on Web service requests and responses as they pass through the pipeline.

Creating a PIPELINE resource definition

We use the CICS supplied PIPELINE configuration file for a SOAP 1.1 handler. The directory contains the CICS supplied pipeline configuration file:

/usr/lpp/cicsts/cicsts32/samples/pipelines/

1. Copy file basicsoap11provider.xml to directory u/cicsrs6/pipelines.
2. Create the pipeline resource definition, as shown in Figure 7-16 on page 183.


```

OVERTYPE TO MODIFY                                CICS RELEASE = 0650
CEDA ALTER PIPeline( TRPIPE01 )
  Pipeline      : TRPIPE01
  Group         : TRADER
  Description    ==> Basic SOAP 1.1 provider pipeline
  SStatus       ==> Enabled          Enabled | Disabled
  Respwait      ==> Deft             Default | 0-9999
  Configfile    ==> /u/cicsrs6/pipelines/basicsoap11provider.xml
  (Mixed Case) ==>
              ==>
              ==>
              ==>
  Shelf         ==> /u/cicsrs6/trader/shelf
  (Mixed Case) ==>
              ==>
              ==>
              ==>
  Wsdir         : /u/cicsrs6/trader/wsbind/provider
+ (Mixed Case) :

                                           SYSID=PAA1 APPLID=SCSCPAA1

PF 1 HELP 2 COM 3 END                      6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL

```

Figure 7-16 Pipeline definition attributes

Creating a TCPIPService resource definition

To access a Web Service that is exposed in CICS, we must tell CICS to listen on a port. The port number that CICS is listening on is defined in a TCPIPService definition. We define a TCPIPService Resource definition in CICS, as shown in Figure 7-17, to tell CICS to receive the inbound HTTP requests on a specific port.

```

OVERTYPE TO MODIFY                                CICS RELEASE = 0650
CEDA ALTER TCpipservice( SOAPPORT )
  TCpipservice  : SOAPPORT
  GROup         : TRADER
  DDescription   ==> CICS WEB SERVICE PORT
  Urm           ==> NONE
  POrtnumber    ==> 05276          1-65535
  SStatus       ==> Open          Open | Closed
  PROtocol      ==> Http          IIop | Http | Eci | User | IPic
  TRansaction    ==> CWXN
  Backlog       ==> 00001        0-32767
  TSqprefix     ==>
  Ipaddress     ==>
  SOrcketclose  ==> No           No | 0-240000 (HHMMSS)
  Maxdatalen    ==> 000032       3-524288
  SECURITY
  SS1           ==> No           Yes | No | Clientauth
  CErTificate   ==>
+ (Mixed Case)

                                           SYSID=PAA1 APPLID=SCSCPAA1

PF 1 HELP 2 COM 3 END                      6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL

```

Figure 7-17 TCP/IP Service definition attributes

Deploying the Web service provider

To expose a CICS program as a Web service, simply installing the PIPELINE definition causes CICS to look for WSBIND files in WSDir directory and to dynamically install WEBSERVICE and URIMAP resources for each Web service.

Deploying the Web service binding file

The Web service binding file that DFHLS2WS creates is deployed into the CICS region dynamically when we install a PIPELINE resource. A PIPELINE scan command is issued, either explicitly through a CEMT PERFORM PIPELINE() SCAN or automatically during a PIPELINE installation. CICS scans the pickup directory to search for Web service binding files, that is for file names with the .wsbind extension. For each binding file that is found, CICS determines whether to install WEBSERVICE and URIMAP resources. When auto-installed in this way, a WEBSERVICE and URIMAP are automatically discarded when the associated PIPELINE is discarded, which makes for easy management of Web services resources in CICS.

To install the TRPIPE01 resource that we just created, type `CEDA INSTALL G(TRADER) PIPELINE(TRPIPE01)`, and press **Enter**. An “INSTALL SUCCESSFUL” message is displayed.

To verify that the PIPELINE properly installed:

1. Use a CEMT INQUIRE PIPELINE command to verify that the PIPELINE was properly installed. A status of ‘Ena’ (Enabled) is displayed.
2. Tab down and place an S to the left of the pipeline entry, and press **Enter**. the pipeline’s details is displayed.
3. If the pipeline does not have a status of ‘Ena’, look in the MSGUSR section of the CICS joblog. The likelihood is that CICS did not find the pipeline configuration file or write to the shelf directory.

Use a CEMT INQUIRE WEBSERVICE command to verify the WEBSERVICE is installed properly.

Use CEMT INQUIRE URIMAP command to look at the URIMAP that was installed when the PIPELINE SCAN was performed.

Publishing the WSDL to clients

There are many ways to make WSDL files available for clients or development teams to discover. For simplicity, we use FTP to copy the file from the host to our workstation.

Testing the Web service in Rational Developer for System z (RDz)

To test the TraderBL Web service with Web services Explorer in RDz, RDz needs the WSDL that was downloaded earlier. We then use the Web services Explorer to invoke the TraderBL Web service in the CICS.

To test the TraderBL Web service in Rational Developer for System z:

1. Start RDz.
2. Open or change to a Resource perspective.
3. Create a General project called Trader.
4. Import traderbl.wsdl into the Trader project.
5. From the Navigator view, expand the Trader project, right-click **traderbl.wsdl**, and from the context menu, select **Web services** → **Test with Web services Explorer**. A Web Browser view opens.

6. Double-click the Web services Explorer tab to maximize its size.
7. In the Navigator pane of the Web services Explorer, click **TRADERBLHTTPSoapBinding**.
8. In the Actions pane, locate the Endpoints section, and verify that the endpoint is `http://wtsc66.itso.ibm.com:5276/trader/traderbl`. Click **Go**.
9. In the **Navigator** pane of the Web services Explorer, click **+** to expand **TRADERBLHTTPSoapBinding**.
10. Select **TRADERBLOperation** (the operation name). An Invoke a WSDL Operation view opens.
11. In the Actions pane, type `Get_Company` in the text box at the field **request_type**, and click **Go**. This causes RDz to invoke the CICS Web service. In the status pane, the Web service response is as shown in Figure 7-18.

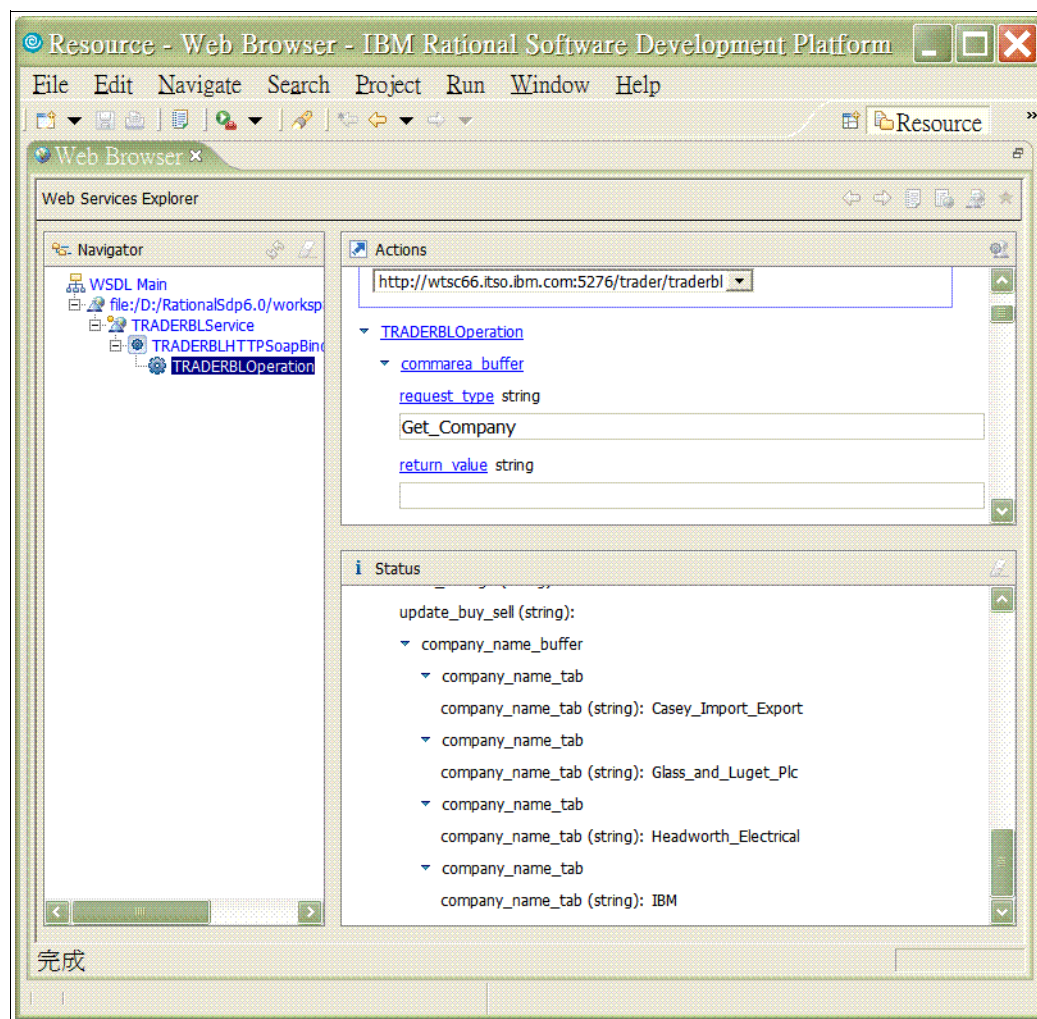


Figure 7-18 RDz Web services Explorer

7.6.2 Migrating TraderPJ to a Web service requester

With TraderBL exposed as a Web service, we now migrate TraderPJ to be a Web service requester. To allow coexistence with the LINK version of program TraderPJ, we develop a new Web service requester program called TraderPW.

Invoking an existing Web service from a new or modified CICS program involves the following steps:

1. Run the CICS Web services assistant to generate the required files:
 - a. Run the program DFHWS2LS.
 - b. Develop the CICS program to invoke the Web service.
2. Set up the CICS infrastructure:
 - a. Create a PIPELINE resource definition.
3. Deploy the Web service requester:
 - a. Deploy the Web service binding file.
 - b. Test the CICS program to invoke the Web service.

Running the CICS Web Service assistant to generate the required files

The DFHWS2LS (Web Service to Language Structure) program is the CICS supplied batch procedure to import a WSDL file and to generate language structures for client applications to use and a WSBIND file to map data between XML and language structures at runtime. DFHWS2LS can handle DOC literal, RPC literal, or wrapped DOC literal forms of WSDL as input. The language structure can be COBOL, PL/I, C, or C++.

We have the CommareaWrapper that was migrated from the COBOL language structure previously, and because the content is the same as the generated COBOL language structure that we will generate, we generate a COBOL language structure and use CommareaWrapper instead.

Running the program DFHWS2LS

The DFHWS2LS utility takes WSDL and generates language structure and WSBind files, as shown in Figure 7-19 on page 187.

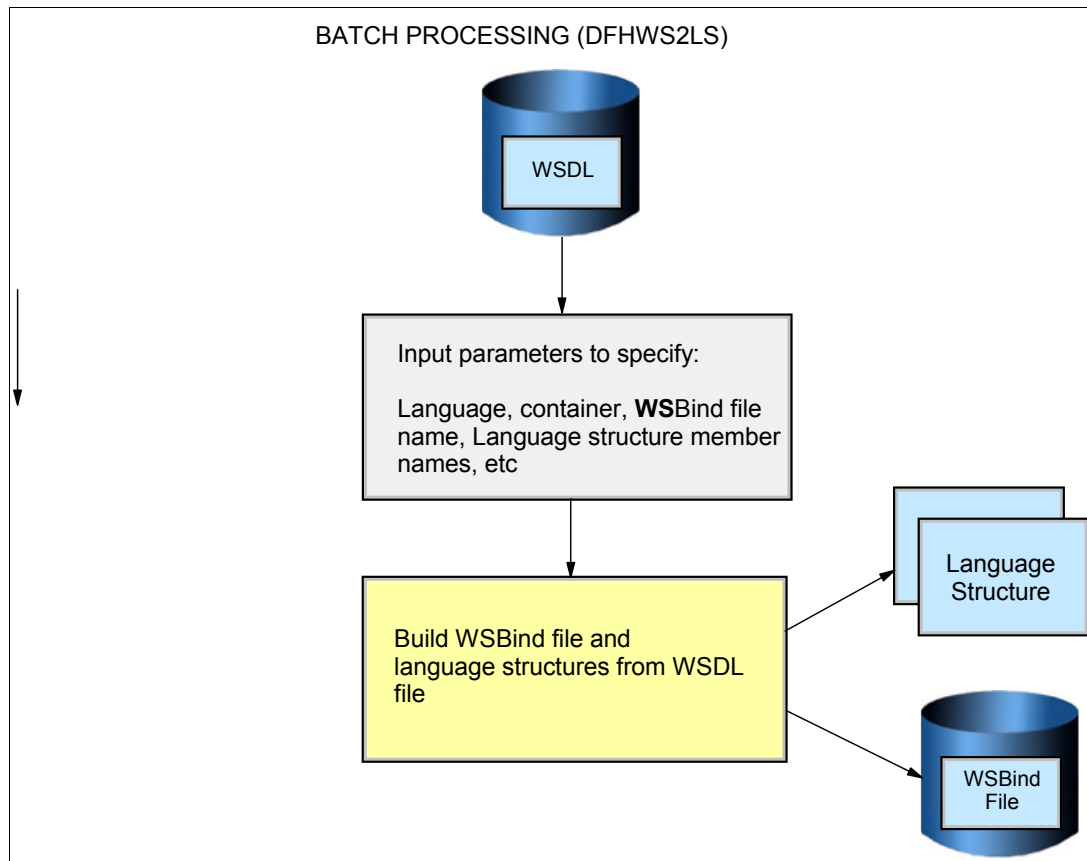


Figure 7-19 DFHWS2LS processing

We must start with a copy of the WSDL file that describes the Web service to be called. In our case, the WSDL file is located on the zFS at:

```
/u/cicsrs6/trader/wsd/traderbl.wsd
```

The CICS supplied sample JCL to run DFHWS2LS is available on the zFS in the file /usr/lpp/cicsts/cicsts32/samples/webservices/JCL/WS2LS.

Note there is no program name, language, or interface supplied because the WEBSERVICE resource does not call a program. Because an URI is not specified, the default is to use the value in the WSDL document, in our case it is `http://wtsc66.itso.ibm.com:5276/trader/traderbl`, which is invoked on the INVOKE WEBSERVICE command. The JCL was customized, as shown in Example 7-39.

Example 7-39 The DFHWS2LS JCL

```
//WS2LS      EXEC DFHWS2LS,
// TMPFILE='ws2ls',
// JAVADIR='/usr/lpp/java/J5.0',
// PATHPREF=' ',
// USSDIR='cicsts32'
//INPUT.SYSUT1 DD *
PDSLIB=//CICSR11.TRADER.COPYLIB
LANG=COBOL
REQMEM=TRADEI
RESPMEM=TRADEO
MAPPING-LEVEL=2.1
```

```
LOGFILE=/u/cicsrs6/trader/wsbinding/dfhws2ls.log
WSBIND=/u/cicsrs6/trader/wsbinding/requester/traderplRequester.wsbinding
WSDL=/u/cicsrs6/trader/wsd1/traderbl.wsd1
BINDING=TRADERBLHTTPSoapBinding
/*
```

Note: Because the WS2LS job is creating two new members in the CICSRS1.TRADER.COPYLIB dataset, if the dataset is in use, the job fails.

View the generated language copybook files. Note that the names of the generated copybooks are the values of TRADEI and TRADEO with a "01" appended to them. This allows DFHWS2LS to generate more than one copybook if the WSDL contains multiple interface definitions. In our case, we do not use the two copybooks; instead, we use the CommareaWrapper.

Developing the CICS program to invoke the Web service

This is the point where a developer writes the CICS program to invoke the Web service. In our case, we use the TraderPL program as base and write a new program TraderPW.

Example 7-40 shows interesting excerpts from TraderPL.

Example 7-40 TraderPW: interesting excerpts from TraderPL

```
private Channel channel;
private Container container;

// Send the data request to TRADERBL
invokeWebService();

/*
 * Invoke webservice to the TRADERBL program with our COMMAREA
 */
private void invokeWebService() {
    final WebService webservice = new WebService();
    webservice.setName("traderplRequester");

    try {
        // Set the UserId value in the COMMAREA
        fCommareaWrapper.setUserId(Task.getTask().getUserID());

        if (channel == null ){
            channel = Task.getTask().createChannel("SERVICE-CHANNEL");
            container = channel.createContainer("DFHWS-DATA");
        }
        container.put(fCommareaWrapper.getBytes());

        webservice.invoke(channel, "TRADERBL0peration");

        fCommareaWrapper.setByteArray(container.get());
    }
}
```

```
        catch (Exception e) { System.err.println(e); e.printStackTrace();
fMessage = "Error - " + e; }
    }
```

Note that the `container.put()` method copies the request data into a `CONTAINER` in the `CHANNEL` (`CHANNEL` and `CONTAINERS` are a new feature in CICS TS Version 3). The `CHANNEL` and `OPERATION` are then passed as a parameter on the `webservice.invoke()` method.

To share HTML templates with `TraderPJ` and `TraderPW`, we add a symbol called `pgmname`, and the default value is set as `traderpj`. The default value make program `TraderPJ` does not need to be changed. The HTML equivalent for this is:

```
<!--#set var=pgmname value=traderpj -->
<FORM action='&pgmname;' method='POST'>
```

Example 7-41 shows the code equivalent for this.

Example 7-41 TraderPW: add program name symbol

```
private String pgmname;

pgmname = Task.getTask().getProgramName();

final String symbolList = "pgmname=" + pgmname + "&" +
    "company1=" + fCommareaWrapper.getCompanyNameTab(0) + "&" +
    "company2=" + fCommareaWrapper.getCompanyNameTab(1) + "&" +
    "company3=" + fCommareaWrapper.getCompanyNameTab(2) + "&" +
    "company4=" + fCommareaWrapper.getCompanyNameTab(3) + "&" +
    "message=" + (fMessage != null ? fMessage : "");
```

Setting up the CICS infrastructure

For a CICS program to make a Web service request over HTTP, a `WEBSERVICE` and a `PIPELINE` resource are needed. A `TCPIP SERVICE` is not needed for a requester Web service call. The client program uses the `WebService invoke method()` method to make the call.

Creating a PIPELINE resource definition

The `PIPELINE` resource that we create refers to a configuration file (`CONFIGFILE`), two `zFS` directories, a `SHELF` directory, and a `WSDIR` directory. We created the directories in “Creating the `zFS` directories” on page 179.

We use the CICS supplied `CONFIGFILE` for a basic SOAP 1.1 requester. The directory contains the CICS supplied pipeline configuration file
`/usr/lpp/cicsts/cicsts32/samples/pipelines/`:

1. Copy the `basicsoap11requester.xml` file to the directory `u/cicrs6/pipelines`.
2. Create the pipeline resource definition, as shown in Figure 7-20 on page 190.

```

OVERTYPE TO MODIFY                                CICS RELEASE = 0650
CEDA Alter Pipeline( TRPIPE02 )
  Pipeline      : TRPIPE02
  Group         : TRADER
  Description    ==> BASIC SOAP 1.1 REQUESTER PIPELINE
  SStatus       ==> Enabled          Enabled | Disabled
  Respwait      ==> Deft             Default | 0-9999
  Configfile    ==> /u/cicsrs6/pipelines/basicsoap11requester.xml
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Shelf         ==> /u/cicsrs6/trader/shelf
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Wsdir         : /u/cicsrs6/trader/wsbind/requester
+ (Mixed Case)  :

                                           SYSID=PAA1 APPLID=SCSCPAA1

PF 1 HELP 2 COM 3 END                      6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL

```

Figure 7-20 Pipeline definition attributes

Deploying the Web service requester

Each CICS Web service request requires at least the following to be installed:

- ▶ A PIPELINE resource that is configured as a requester pipeline
- ▶ A WEBSERVICE resource

Installing a PIPELINE definition causes CICS to look for WSBind files in the WSDir directory, and then CICS dynamically installs a WEBSERVICE for each Web service.

Deploying the Web service binding file

To install the TRPIPE02 resource that we just created:

1. Type CEDA INSTALL G(TRADER) PIPELINE(TRPIPE02), and press **Enter**. An “INSTALL SUCCESSFUL” message is displayed.
2. Use the PERFORM PIPELINE() SCAN command as an alternative way to pick up the WSBind file automatically, and dynamically create the WEBSERVICE resource. A URIMAP is not created for a requester Web service call.

Verifying PIPELINE and WEBSERVICE installation:

Use the CEMT INQUIRE PIPELINE command to verify that the PIPELINE is installed properly.

Use the CEMT INQUIRE WEBSERVICE command to verify that the WEBSERVICE is installed properly.

Testing the CICS program to invoke the Web service

In this section, we tell you how to install and run TraderPW.

Installing TraderPW

The source code and compiled classes for this example are in Trader-2.jar. Following the instructions in Chapter 3, “Setting up CICS to run Java applications” on page 37, make sure this jar file is added to the CICS Java classpath.

The new CICS resources that must be defined alongside the existing definitions in the TRADER group are:

- ▶ PROGRAM(TRADERPW)
CONCURRENCY(Threadsafe)
JVM(Yes)
JVMCLASS(com.ibm.itso.sg245275.trader.TraderPW)

Running TraderPW

Similar to “Running TraderPL” on page 158, open a Web browser and type in the URL:

<http://wtsc66.itso.ibm.com:5275/cics/cwba/traderpw>

Figure 7-21 is displayed.

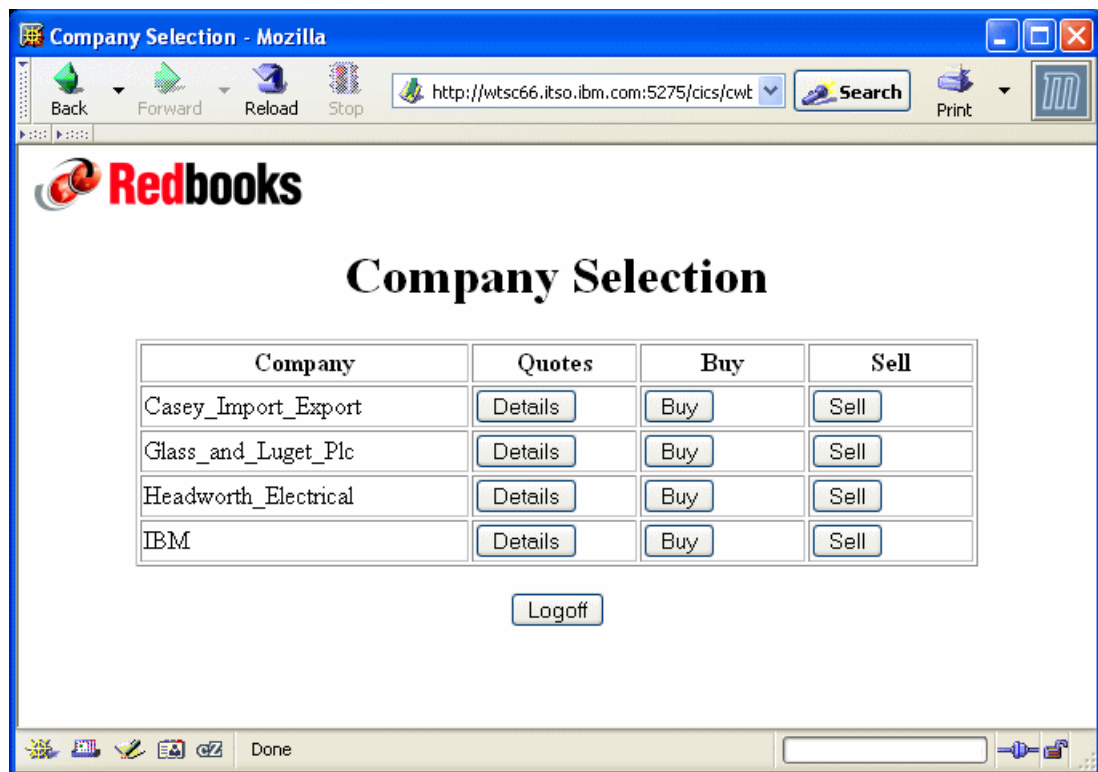


Figure 7-21 The company selection window



Problem determination and debugging

In this chapter, we describe some tools and techniques that can help you discover and fix problems in your application or configuration.

In the first section, we describe some of the tools and techniques for problem determination. In this section, we summarize the most common problems and the source of information that can help you identify and solve the problems. We introduce some facilities that system programmers are familiar with, including various of dumps, traces, and the tools that can help to analyze them.

In the second section, we describe using Rational Developer for System z to perform interactive debugging to help an application developer find and correct problems in the Java application quickly and easily.

8.1 Debugging and problem determination

There are times when your CICS Java application does not display the expected or desired behavior. Your application might:

- ▶ Not start at all.
- ▶ Produce incorrect output.
- ▶ Enter a loop.
- ▶ Wait indefinitely for another resource.
- ▶ Perform badly.
- ▶ Result in a system message, abend, or dump.

There might be a problem with the application itself that you can discover and correct yourself. The configuration can be in error, which prevents the application from executing correctly. Fixing the configuration allows the application to execute as expected.

We discuss some tools and techniques to help you resolve these issues.

You might encounter a problem with the SDK, Language Environment, or CICS, which requires assistance from IBM to resolve. One aim of this section is to describe how to gather the documentation that IBM is likely to require and how to identify which area of IBM support is required.

CICS provides an extensive set of problem determination aids for systems and application programmers. Many of the usual sources of CICS diagnostic information, such as abend codes, error messages, dumps, and trace, are useful in resolving a CICS Java problem.

However, there are extra considerations when diagnosing a problem in a CICS Java application. Because of the nature of the JVM, you might need to look at diagnostic information from UNIX System Services, Language Environment, and IBM SDK for z/OS.

The following Web site contains useful hints and tips for IBM SDK for z/OS problem solving. You can use the information at this site along with other sources of information, such as this IBM Redbooks publication and *CICS Problem Determination Guide*, SC34-6239, to develop a strategy for carrying out problem determination for CICS Java applications.

<http://www-1.ibm.com/servers/eserver/zseries/software/java/javafaq.html>

The JVMs own diagnostic tools and interfaces that give you detailed information about what is happening within the JVM than CICS can because CICS is unaware of many of the activities within a JVM. You can download the diagnostics guide for the IBM SDK from the following Web site:

<http://www-128.ibm.com/developerworks/java/jdk/diagnosis/>

8.1.1 First considerations

Before looking in detail at some of the diagnostic tools, we describe some points you must consider if you encounter a problem with CICS Java. There might be a straightforward explanation for the problem, for example, the wrong JVM profile or properties file might have changed or a typing error might have occurred when entering a path to a file or directory. You can save a lot of time by carrying out these initial checks.

Which version of Java are you using

IBM provides regular service updates to the SDK. We recommend that you install the latest service level to take advantage of improved service, performance, and stability. More

information about the PTF numbers, build date, and availability are on the Java Service Summary Web site:

<http://www-03.ibm.com/servers/eserver/zseries/software/java/services/j5servsum31.html>

Table 8-1 shows parts of the SDK 5 service releases at the time of writing.

Table 8-1 SDK 5 service releases

Service release	PTF number	Build date
SR8a	UK39047	August. 11, 2008
SR8	UK38281	July.10, 2008
SR7	UK34636	March.15, 2008

Tip: The system property `java.full version` tells you the build level of the SDK you are using, which is written to the event log along with the other system properties if event logging is enabled.

If you specify the system property `com.ibm.cics.showJavaVersion=true`, CICS writes message DFHSJ0901 to CSMT when the JVM initializes. The message text contains the Java version (including the build date) used for that invocation of the JVM.

Is the JVM failing to start

Is the JVM failing to start:

- ▶ Does the RACF profile for the CICS region user ID have an OMVS segment?
- ▶ Does the directory specified for `WORK_DIR` exist and does the CICS region user ID have read and write permissions to it?
- ▶ Are the path names to the Java directory correct and in sync on the `JAVA_HOME` and `LIBPATH` options?
- ▶ Does the CICS region user ID have read permission to all directories on the path?
- ▶ Is the value specified for the `CICS_DIRECTORY` correct?

Which JVM mode is in use

Which JVM mode is in use:

- ▶ Is the JVM using the expected JVM profile?
- ▶ Is the JVM running in continuous mode?
- ▶ Is the JVM using the Shared Class Cache?
- ▶ Is the JVM running in the correct key (CICS or USER)?

Has the JVM abended

If you see the following message in the CICS log, the JVM (or some native code running under the JVM) abended, but all CICS knows is that the call to `CEEPIPI` returned an error code.

```
DFHSJ0203 A call to CEEPIPI with function code CALL_SUB has
failed. (Return code was - X'0000001C')
```

You will require assistance from IBM in resolving this type of problem. You must gather any diagnostics that the JVM produces. There might be a `JAVADUMP` in the working directory.

8.2 Common problems

In this section, we provide some common error messages that you are likely to encounter.

8.2.1 Abend AJ04

You receive abend AJ04 when CICS cannot find your program's main method. This is a common error. Example 8-1 shows an example of an abend AJ04.

Example 8-1 Most common abend AJ04

```
DFHAC2206 04:25:32 A6POC3C4 Transaction HEL0 failed with abend AJ04. Updates to
local recoverable resources backed out.
```

The abend AJ04 is typically caused by:

- ▶ Check the name of the main class on the program definition (CEMT INQUIRE PROGRAM) and make sure that it is spelled correctly.
- ▶ Double-check the main class name. It is easy to miss a typo.
- ▶ Check the classpath settings in the JVM profile.
- ▶ Make sure your program definition uses the correct JVM profile.
- ▶ Check that your main class does actually have a public static void main(String []) or public static void main(CommAreaHolder) method.
- ▶ Make sure CICS has access to the directories and files in the classpath.
- ▶ When deploying the Java bytecode to the mainframe, make sure it is transmitted in binary mode. FTP and Rational Developer for System z can perform character set conversions if the default setting is overridden. Make sure no such conversion takes place for .class and .jar files; otherwise, you will probably see the message, in Example 8-2, in the log.

Example 8-2 Bad magic number is usually caused by improper conversion during the transmission

```
DFHSJ0904 11/12/2008 04:14:54 A6POC3C4 CICSUSER E011 HEL0 HELOPROG Exception
java.lang.ClassFormatError:
(com/ibm/itso/sg245275/HelloWorld) bad magic number at offset=0 occurred creating
object reference for class com.ibm.itso.sg245275.HelloWorld.
DFHDU0203I 11/12/2008 04:14:54 A6POC3C4 A transaction dump was taken for dumpcode:
AJ04, Dumpid: 1/0003.
DFHAC2236 11/12/2008 04:14:55 A6POC3C4 Transaction HEL0 abend AJ04 in program
HELOPROG term E011. Updates to local recoverable resources will be backed out.
```

- ▶ When using file system export from Rational Developer for System z, it is a common mistake to use the “Create directory structure for files” and “Create only selected directories” options inconsistently (especially when using the latter). Check that your .class files ended up on the zFS file system where you want them.

8.2.2 Incorrect output or behavior

It is also normal to find out that your program is not generating expected messages or behavior. In this case, the reason can vary. You might need to check the stdout and stderr to see if any exception is thrown by your problem. If there is nothing obvious in the log, you might need to turn on the trace or take some dump to analyze the detail reason. Or you might want to try to debug your program interactively to see what is happening. See 8.4, “Interactive debugging” on page 212 for details.

8.2.3 No response

In this case, your program might loop infinitely or wait for some resource to become available, possibly a deadlock. Usually the infinite looping takes up high CPU times although the resource wait and deadlock takes up almost no CPU time at all. You can distinguish it by looking at the CPU time of the CICS region.

In the case of deadlocks, you probably need to use the Jvareg tool to diagnose the lock status and the resources that cause the deadlock. We explain it later.

If the programing is in infinite looping, you can analyze what the thread is doing by taking a console-initiated dump (SVC dump), then use the dump viewer to see what is happening and why it loops infinitely. See more detail in *SDK Diagnostics Guide*, SC34-6650.

8.2.4 OutOfMemoryError

There are chances that your application reports OutOfMemoryError exception. Typically the JVM reports this error when the heap storage is exhausted and cannot allocate new objects, or when a JVM's attempt to call malloc() is failed. The possible causes are:

- ▶ Insufficient heap size: Your application might need more memory than defined.
- ▶ The program design problem: There might be some memory leak.
- ▶ JVM fails to get more memory by malloc().

If it is because the program requires more memory than defined, using the parameter -Xmx in the JVM profile, you must increase the maximum heap size to solve this problem.

Java uses references to keep track of the live objects in memory. An Object with at least one reference to it from a live thread or a static value is considered a live object and is not removed from memory during a garbage collection cycle.

To make an object eligible for garbage collection, the reference to that object must be set to null or must be out of scope. It is not necessary to explicitly set every reference to null, so for example, an object referenced only by a method local variable becomes eligible for garbage collection when the method ends and the variable goes out of scope, but memory leaks are caused by failure to remove references to unused objects. If you are in doubt about whether an object reference is causing a memory leak, it is always safer to set that reference to null after it is no longer required. Failing to remove references to objects that are no longer required results in unnecessary objects that take up more and more heap and finally cause an OutOfMemoryError exception to occur when the heap fills.

If it is that JVM calls to malloc() failed, you will likely get an associated error code. In this case, you can use the following useful rules to estimate the spaces required.

Since $EDSALIM + MVS\ STORAGE = REGION\ SIZE$
REGION SIZE is defined in CICS JCL (allocated up front)
EDSALIM is SIT parameter in CICS JCL (allocated up front)
MVS STORAGE is used for JVMs (allocated when required)
Max no. of JVMs in a region set using MAXJVMTCS SIT parameter
Size of JVM approx $25Mb * Xmx$ (Xmx is max JVM heap size set in DFHJVMPR)
Now do the maths:
If $EDSALIM + (MAXJVMTCS * (25Mb + Xmx)) > REGION\ SIZE$,
then Java may run out of storage.

To solve the OutOfMemoryError problem, first identify if it is the configuration problem or the application design problem. If it is due to the wrong configuration, just increase the maximum

heap size in the JVM profile to solve the problem. If it is the application design problem, you probably need to:

- ▶ Review your code and correct the way for managing the object collections
- ▶ Enable the heapdump to analyze what is in the heap when the error occurred, which helps you to identify the problem quickly, which we explain in 8.3.2, “Heapdump” on page 203.
- ▶ Switch on -verbose:gc output, which causes the JVM to print out a message before and after a garbage collection cycle. If the total available space on the heap decreases overtime after each GC cycle, there must be some object leak. See more detail in 8.3.3, “Monitoring garbage collection cycles” on page 204.

8.2.5 Performance is not good

The performance problem is always difficult to solve. Here are some possible aspects that might affect the performance of the application:

- ▶ Insufficient CPU power, memory, or I/O capability
- ▶ Improper configuration for the heap size
- ▶ Using Just-In-Time (JIT) compilation
- ▶ Application design problems, for example, poor algorithms, and so on

Chapter 5, “Writing Java 5 applications for CICS” on page 75 offers some general guidance that can help you to avoid some application performance issues.’

Your program might still run slowly even if you do extensive tuning. The hardware environment is still not enough to handle the workload of your application. The only way to solve it is to perform a hardware upgrade to remove the bottleneck.

A poorly chosen of the heap size can also cause the degrade of the performance because the increasing frequency of the garbage collection cycle adds considerable overhead to the system, which might affect your application. To know how often the garbage collection occur, So it is important to select a optimized heap size for your JVM. To see how to select a proper heap size for your program, refer to Chapter 2 How to do heap sizing in *SDK Diagnostics Guide*, SC34-6650.

Use of Just-In-Time compiler can increase the performance of the application. JIT compiler compiles selected methods bytecode to native machine code to avoid the process of interpreting the bytecodes each time it runs, which allow the speed of Java program to approach that of a native program. The JIT compiler is enabled by default, you might want to review the configuration if you hit some performance problem.

You can learn a lot about your Java application by using the HPROF profiling agent to identify if it is the design lead to a poor performance. HPROF is a demonstration profiler that is shipped with the IBM SDK that uses the JVMTI to collect and record information about Java execution. Use it to work out which parts of a program are using the most memory or processor time. For more information about HPROF, refer to the following Web site:

<http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>

8.2.6 Problems caused by static values in continuous JVMs

Static values in Java programs persist between uses of the JVM where a continuous JVM is used. If these values are changed by one transaction that is running in the JVM, subsequent transactions running in that JVM pick up this modified value. These problems are difficult to recreate in test environments because of the interaction of different programs running in the JVM. Guidance on how to avoid these problems is in 2.3.2, “Continuous JVM” on page 21.

The CICS JVM Application Isolation Utility tool that is provided with CICS can help you to check your usage of static fields and classes, which helps you identify these problems, and we describe this tool in 2.4, “Analyzing programs for use in a continuous JVM” on page 23.

8.3 Where to look for diagnostic information

You can look in many places to gather all of the diagnostic information to solve a problem. The various components that are required to provide CICS Java support writes diagnostics to different locations.

You might find diagnostic information in familiar places:

- ▶ MVS console messages
- ▶ CICS messages written to TD queues or terminal end user
- ▶ CICS transaction abend codes
- ▶ CICS trace

You can use standard problem determination techniques in this case, for example, you can look in *CICS Messages And Codes*, GC34-6442, for an explanation of a CICS message or transaction abend code. A suggested course of action that you can take to resolve the problem might also be included with the explanation. However, for a fault in the JVM you must inspect additional diagnostic information. You can get some of this information by default; however, you might have to specify JVM options to obtain the information you need to resolve a problem.

The extra information can come from any of these items:

- ▶ Java stack traces when exceptions are thrown
- ▶ Javadumps:
 - Transaction dump: An unformatted dump requested by the MVS IEATDUMP service. You can post-process this dump with Interactive Problem Control System (IPCS).
 - CEEDUMP: Formatted application level dump, requested by the Language Environment service CEE3DMP.
 - JAVADUMP: Formatted internal state data produced by the IBM JVM.
- ▶ Heapdump: JVM generates a heap dump when the heap is exhausted or by user request. It contains all of the live objects in the heap when the dump is taken
- ▶ Debugging messages written to stderr or stdout (for example, the output from the JVM when switches, such as -verbose:gc, -verbose, or -Xtgc are used)
- ▶ Binary or formatted trace data from the JVM internal high-performance trace
- ▶ SVC dumps that the MVS Console DUMP command obtains (typically for loops or hangs)
- ▶ Trace data from other products or components (for example, Language Environment traces or the Component trace for z/OS UNIX)

In the next sections, we provide more information about the diagnostics that might not be familiar to those of you who are from a traditional CICS background.

8.3.1 Javadumps

The JVM has code to create Javadumps when the JVM terminates or crashes unexpectedly. You can control the code by using environment variables and runtime switches. A Javadump is a file that attempts to summarize the state of the JVM at the instant the signal occurred.

The crash might be the result of an error in JVM code or JNI code. If the error is in JVM code or JNI code supplied by IBM, you must contact IBM for assistance. The information in this section helps you to identify the source of the error and to gather the correct documentation for IBM support.

On z/OS, you control the behavior of the Javdump using the `JAVA_DUMP_OPTS` environment variable. You can specify this variable in your JVM profile.

If you do not specify `JAVA_DUMP_OPTS` in your profile, CICS uses the following default:

```
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
```

If a CICS JVM crashes, you receive a formatted Javdump, a Language Environment dump, and an unformatted transaction dump.

You can find the full syntax for `JAVA_DUMP_OPTS` on z/OS in *SDK Diagnostics Guide*, SC34-6358. You should not need to change the dump options unless advised to do so by IBM support.

If the JVM crashes, a JAVADUMP, Language Environment CEEDUMP and Java transaction dump is produced. Any one of the dumps produced might be enough on its own to find the cause of the problem.

JAVADUMP

A JAVADUMP is a formatted text file that provides information about the JVM at the time of the failure. The JVM checks each of the following locations for existence and write-permission and stores the Javdump in the first one available. You must have enough free disk space (possibly up to 2.5 MB) for the Javdump file to be written correctly:

- ▶ The location specified by the `_CEE_DMPTARG` environment variable, if set
- ▶ The current working directory of the JVM processes
- ▶ The location specified by the `TMPDIR` environment variable, if set
- ▶ The `/tmp` directory

The file name is `JAVADUMP.yyyymmdd.hhmmss.&PID.txt`, where PID is the process ID (for example, `JAVADUMP.20050330.074905.txt`).

Each line of the JAVADUMP starts with a tag. You can use this tag to parse the JAVADUMP.

There is a lot of information contained in a JAVADUMP. There is more information about interpreting a JAVADUMP in *SDK Diagnostics Guide*, SC34-6358. We provide some of the information that you can use to obtain from the JAVADUMP here.

At the top of the file is information, such as:

- ▶ The date and time of the dump
- ▶ The build level of the SDK
- ▶ The OS level
- ▶ The signal that caused the dump to be taken (In Example 8-3 on page 201, a SIGSEGV (signal 11) was received.)

Example 8-3 on page 201 shows the first two sections of a JAVADUMP. Each line of the Javdump starts with a tag that can be up to 15 characters long. You can use this metadata to parse and perform simple analysis of the contents of a Javdump.

Example 8-3 JAVADUMP title and operating environment sections

```
NULL -----
OSECTION  TITLE subcomponent dump routine
NULL -----
1TISIGINFO signal 11 received
1TIDATETIME Date:          2005/03/30 at 07:49:05
1TIFILENAME Javacore filename: /tmp/JAVADUMP.20050330.074905.16778293.txt
NULL -----
OSECTION  XHPI subcomponent dump routine
NULL -----
1XHSIGRECV SIGSEGV received at fc1363d8 in (unknown Module)
1XHTIME    Wed Mar 30 07:49:05 2005
1XHFULLVERSION JNI J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm142-20040917
NULL
1XHOPENV   Operating Environment
NULL -----
2XHHOSTNAME Host          : mlp1:10.0.32.42
2XHOSLEVEL  OS Level     : z/OS V01 R05.00 Machine 2084 Node MLP1
2XHCPU      Processors -
3XHCPUARCH  Architecture : (not implemented)
3XHNUMCPUS  How Many    : (not implemented)
3XHCPUSENABLED Enabled    : 4
NULL
```

You can find the environment variables for the JVM by searching for the tag 2XHENVVARS in the JAVADUMP, which shows the options, such as CLASSPATH, CICS_DIRECTORY, and JVMPROPS, that are set in the JVM profile.

The section “Current Thread Details” contains information that helps you identify the failing CICS task and program at the time of the failure. Search for the tag 1XHTHDDetails to find this section. The CICS program name, task number, and transaction identifier of the failing task are on the line with the tag 2XHCURRENTTHD, as shown in see Example 8-4.

Example 8-4 Current thread information from JAVADUMP

```
2XHCURRENTTHD "DFJIIIRP.TASK14008.CIRP" (sys_thread_t:47b59290)
```

You find stack traces throughout the JAVADUMP by searching for the tag 3XHNATIVESTACK. Each visible stack frame contains the name and library of the function that is involved (this information is not always complete). The failing function is immediately before the Language Environment error handling routine CEEHRNUH in the stack.

Example 8-5 on page 202 shows the stack from the CEEPIPI call to invoke the JVM to the failure and Language Environment error handling. In this example, the failing function is buildPinnedFreeList in gc_free.c.

Example 8-5 Native stack in JAVADUMP

CEEHDSP	CEEHDSP	
CEEHRNUH	CEEHRNUH	
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/st/msc/gc_free.c	buildPinnedFreeList	1406
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/st/msc/gc_mwmain.c	gc0_locked	4353
/u/sovbld/cm142/cm142-20040917/src/jvm/pfm/st/msc/gc_md.c	gc_locked	74
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/st/msc/gc_mwmain.c	gc0	5073
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/st/msc/gc_mwmain.c	gcMiddlewareHeap	2000
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/st/msc/gc_main.c	gc	699
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/ci/jvm.c	JVM_GC	2117
/u/sovbld/cm142/cm142-20040917/src/java/sov/Runtime.c	Java_java_lang_Runtime_gc	74
/u/sovbld/cm142/cm142-20040917/obj/mvs390_oe_2/jvm/sov/xe/c_mmi/custom_invokers.c	mmipInvoke_V_V	816
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/xe/c_mmi/mmi_invoke_cmml.c	mmipInvokeLazyJniMethod	650
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/xe/c_mmi/mmi_execute.c	mmipExecuteJava	2478
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/xe/common/run.c	xeRunJniMethod	1085
/u/sovbld/cm142/cm142-20040917/src/jvm/sov/ci/jni.c	jni_CallStaticVoidMethod	507
	sjcscall	
	dfhsjcs1	
	@@FECB	
	CEEVROND	
CEEPIPI	CEEPIPI	

CEEDUMP

The CEEDUMP is produced after any SYSDUMP processing but before a JAVADUMP is produced. A CEEDUMP is a formatted summary system dump that shows stack traces for each thread that is in the JVM process together with register information and a short dump of storage that pertains to each register. The CEEDUMP is written to the CESE transient data queue.

Use the traceback information in the CEEDUMP to find the source of the problem. The entry immediately before the entry for CEEHDSP shows the problem program. CEEHDSP is the Language Environment module that is responsible for scheduling the CEEDUMP.

Example 8-6 shows traceback data from a CEEDUMP.

Example 8-6 CEEDUMP traceback

TRACEBACK:											
DSA ADDR	PROGRAM UNIT	PU ADDR	PU OFFSET	ENTRY	E ADDR	E OFFSET	STATEMENT	LOAD MOD	SERVICE	STATUS	
2E63E178	/u/sovbld/hm131s/hm131s-20020723/src/hpi/pfm/threads_utils.c	2EB098A0	+000006AA	ThreadUtils_CoreDump	2EB098A0	+000006AA	1662 *PATHNAM	h020723	CALL		
2E63E068	/u/sovbld/hm131s/hm131s-20020723/src/hpi/pfm/interrupt_md.c	2EAEC108	+000004B0	userSignalHandler	2EAEC108	+000004B0	376 *PATHNAM	h020723	CALL		
2E63DFB8	/u/sovbld/hm131s/hm131s-20020723/src/hpi/pfm/interrupt_md.c	2EAEC670	+000000B6	intrDispatch	2EAEC670	+000000B6	642 *PATHNAM	h020723	CALL		

2E63DF00		2DED9938	+0000001A	@GETFN	2DED9890	+000000C2	CEEEV003	CALL
2E63DE80	CEEVASTK	2DCCC420	+00000126	CEEVASTK	2DCCC420	+00000126	CEEPLPKA	CALL
2E32BCC0		2DFF2B80	+00000736	__zerros	2DFF2B80	+00000736	CEEEV003	DRIVER9 CALL
2E329128	CEEHDSP	2DC16848	+00000C0C	CEEHDSP	2DC16848	+00000C0C	CEEPLPKA	CALL
2E3286A8	/u/sovbld/hm131s/hm131s-20020723/src/zip/sov/ZipEntry.c							
		31F6EB38	+00000252	Java_java_util_zip_ZipEntry_initFields	31F6EB38	+00000252	99	*PATHNAM h020723
EXCEPTION								
2E3285C8	java/util/zip/ZipEntry.java							
		32CEE95C	+000000E4	java/util/zip/ZipEntry.initFields(J)V	32CEE95C	+000000E4	0	CALL

You can change the level of information that the Language Environment produces when a JVM crash occurs by overriding the TRAP and TERMTHDACT runtime options in DFHJVMRO. We recommend that you do not override these values unless IBM advises you to. The default values that CICS set for the enclave in which the JVM runs are TERMTHDACT(DUMP,,96) and TRAP(ON,NOSPIE).

SYSDUMP

A sysdump is captured using the MVS IEATDUMP service, which produces a data set that contains an unformatted transaction dump. You use MVS Interactive Problem Control System (IPCS) to format the dump.

The default name for the data set that is produced is &userid.SYSTDUMP.&date.T&time. You can use the environment variable JAVA_DUMP_TDUMP_PATTERN to set your own name.

Messages appear in stderr when the JVM requests a sysdump. Example 8-7 shows the messages that we saw. You can see that we use JAVA_DUMP_TDUMP_PATTERN to set our own name for the dump data set.

Example 8-7 Messages in stderr when a dump is captured

```
JVMDBG217: Dump Handler is Processing Signal 4 - Please Wait.
JVMHP002: JVM requesting System Transaction Dump
JVMHP012: System Transaction Dump written to PLOWMAN.JVM.TDUMP.COMKZCES.D050419.T231519
JVMDBG303: JVM Requesting Java core file
JVMDBG304: Java core file written to
/u/plowman/cics630/workdir/JAVADUMP.20050419.231606.83886118.txt
JVMDBG215: Dump Handler has Processed Exception Signal 4.
```

You use IPCS to format the dump. VERBX LEDATA 'CEEDUMP' formats the Language Environment traceback information. You can also use the appropriate CICS formatting exit to format CICS data, such as the internal trace table. *CICS Operations & Utilities Guide*, SC34-6431, tells you how to use IPCS to format CICS dumps.

8.3.2 Heapdump

Heapdump is a dump file that the JVM generates of all the live objects on the Java heap that running applications are using. The heapdump is stored in a Portable Heap Dump (phd) file, which is a compressed binary file. You can use various tools to analyze the heapdump to know what is on the heap, what is using the most memory, and why the garbage collection cannot recycle it.

By default, JVM automatically generates heapdump when the heap is exhausted. You can also change this behavior by using the -Xdump:heap parameter in the JVM profile.

There are tools that can help you to understand the non-readable output heapdump. The preferred heapdump analysis tool is Memory Dump Diagnostic for Java (MDD4J). Figure 8-1 shows how to use MDD4J to analyze the heapdump. You can find it in IBM Support Assistant V4.0 Tools, at the following Web site:

<http://www-01.ibm.com/software/support/isa/isa40/tools.html>

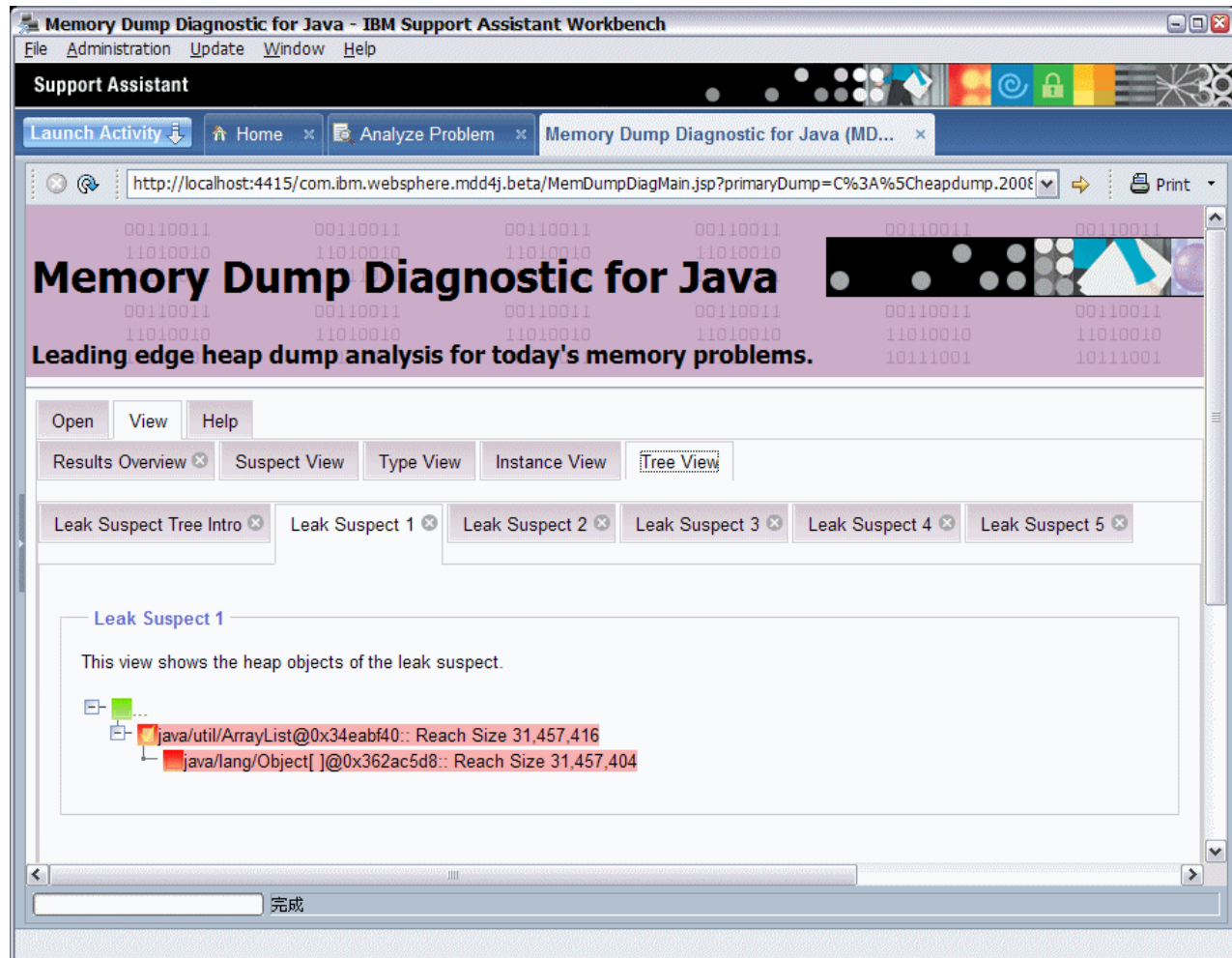


Figure 8-1 Using MDD4J to analyze the heapdump

There are also other utilities that can help you to analyze the heapdump. You can find more information at:

<http://www-01.ibm.com/support/docview.wss?uid=swg24009436>

Of course you still have alternatives. You can also get a classic text-formatted heapdump by specifying the `-Xdump:heap:opts=CLASSIC` parameter in the JVM profile, which gives you a human-readable view of the heap. For more information about how to understand the classic heapdump, refer to *SDK Diagnostics Guide*, SC34-6650.

8.3.3 Monitoring garbage collection cycles

The JVM intermittently performs garbage collection. Although Java does not specify exactly when garbage collection occurs, there are circumstances that make garbage collection likely. If the heap free space drops below a given threshold, a garbage collection is likely. Typically if this cycle fails to free sufficient heap storage, the JVM might request more heap storage

from z/OS (up to the maximum heap size). If the program invokes `System.gc()`, this can cause the JVM to run a garbage collection cycle at some point, but this is not guaranteed. The JVM also might run garbage collection at regular (or not so regular) intervals.

Because garbage collection stops all processing and can take a significant period of time to complete, it can cause performance problems.

There are times that the garbage collection can result in problems for your application. Sometimes the garbage collection occurs too often, which brings a huge overhead. Sometimes it brings too much pause time. And sometimes it just does not work as expected, and you still get an `OutOfMemoryError` exception. In such cases, you might want to see what is happening inside JVM.

The first tool to use to diagnose the garbage collection problems is the verbose logging. You can enable it by specifying the `-verbose:gc` parameter in the JVM profile. The output is written to `syserr` by default. Example 8-8 is a sample of the output when a global collection is performed.

Example 8-8 Sample -verbose:gc output during the global garbage collection

```
<gc type="global" id="5" totalid="5" intervalms="18.880">
<compaction movecount="9282" movebytes="508064" reason="forced compaction" />
<expansion type="tenured" amount="1048576" newsize="3145728" timetaken="0.011"
reason="insufficient free space following gc" />
<refs_cleared soft="0" weak="0" phantom="0" />
<finalization objectsqueued="0" />
<timesms mark="7.544" sweep="0.088" compact="9.992" total="17.737" />
<tenured freebytes="1567256" totalbytes="3145728" percent="49" >
  <soa freebytes="1441816" totalbytes="3020288" percent="47" />
  <loa freebytes="125440" totalbytes="125440" percent="100" />
</tenured>
</gc>
```

You can use the tool *The IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer*, Figure 8-2 on page 206, to get a nice graph about heap usage and garbage collection results by using the output of `-verbose:gc` as the input file. It is also included in the IBM Support Assistant V4.0 Tools, which are found at:

<http://www-01.ibm.com/software/support/isa/isa40/tools.html>

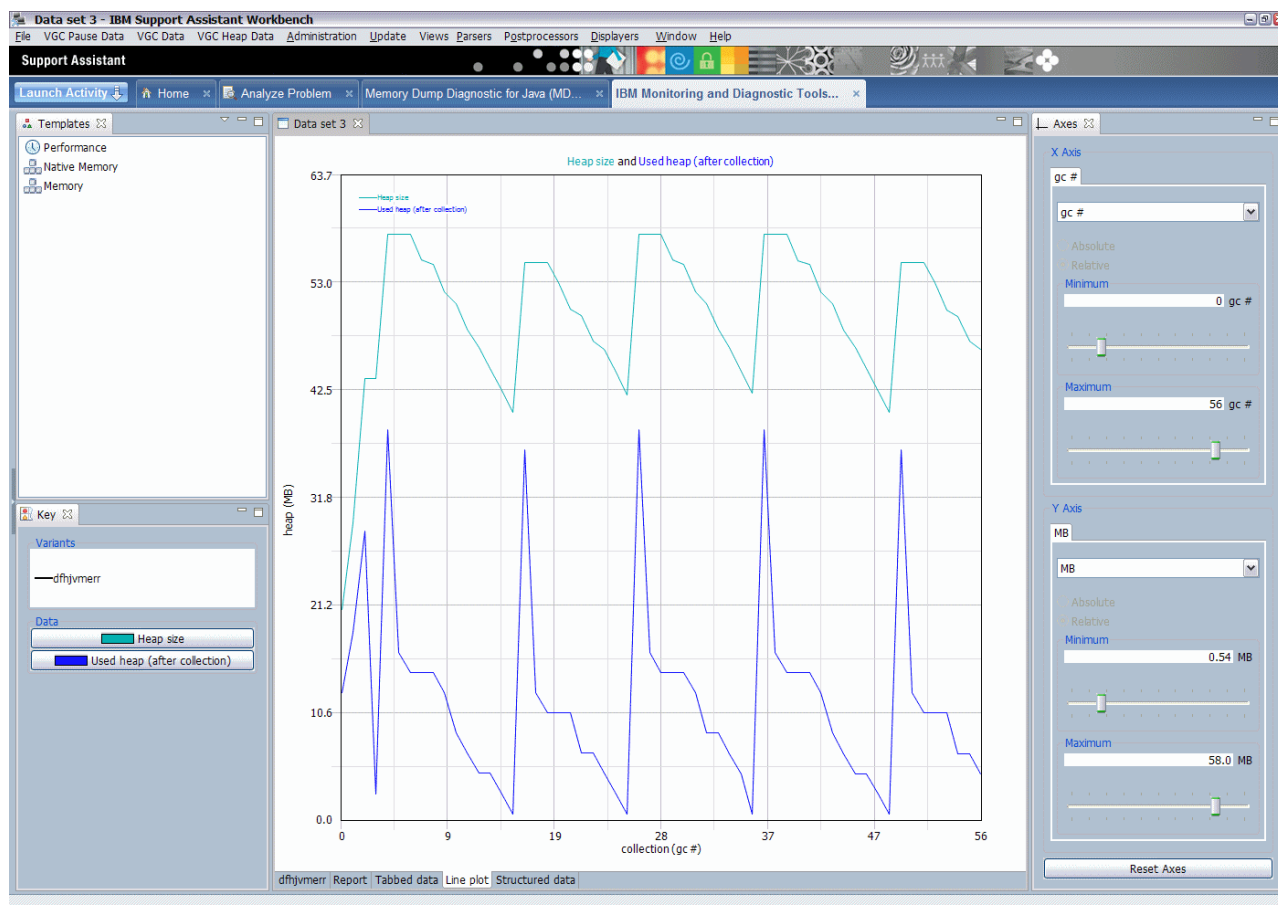


Figure 8-2 The IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer

You can also get more detailed information by enabling one or more trace garbage collectors (TGC) by specifying the `-Xtgc` parameter in the JVM profile. For more information, refer to *SDK Diagnostics Guide*, SC34-6650.

8.3.4 JVM stdout and stderr

The JVM and the applications that it is running might write data to stdout and stderr. The JVM writes informational and error messages to stdout and stderr, for example, if the JVM abends, messages are written to stderr. The application itself might write messages to stdout and stderr, which might provide important clues in resolving a problem.

The location of these files is specified in the JVM profile using the options `STDOUT` and `STDERR`. Look in the JVM profile to find where stdout and stderr are written.

The default names for stdout and stderr are `dfhjvmout` and `dfhjvmerr`, respectively. By default they are created in the `WORK_DIR` directory. You can use your own file name or specify a full path name. Remember, you can also use the symbol `&APPLID` in the JVM profile to substitute the CICS region applid in any name at runtime.

You can further modify the name of the stdout and stderr files by using the **-generate** option in the JVM profile (for example, `STDOUT=dfhjvmerr -generate`. There must be a space before the `-generate`), which tells CICS to generate a specific file for the JVM by appending the following information to the specified file name:

region	The applid of the CICS region
time	The current time in the form yyddhhmmss
.txt	A literal string suffix to indicate that the file contains readable data

An example of a generated name is `dfhjvmout.SCSCPJA7.05101150145.txt`.

We do not recommend that you use the `-generate` option in a production environment because it can affect JVM performance. However, you might want to use it in a test environment to keep output from different test cases separate.

Note: The `USEROUTPUTCLASS` option in the JVM profile specifies the name of a class that is used to redirect output from the JVM. If this option is set in the JVM profile, you must know how the specified class deals with output from the JVM. *Java Applications in CICS*, SC34-6238, contains more information about redirecting JVM output using these classes.

8.3.5 JVM method tracing

Using the powerful tool method trace, you can trace method entry and exit in any Java code. You do not add any hooks or calls to existing code. Method trace provides a complete (and potentially large) diagnosis of code paths inside your application and also inside the system classes. Use wild cards and filtering to control method trace, so that you can focus on the sections of code that interest you.

You control method trace by specifying system properties, which you specify in the JVM profile. All of the method trace properties are of the format `ibm.dg.trc.<something>`. The set of these properties is quite large and is fully described in *SDK Diagnostics Guide*, SC34-6650.

If you want method trace to format, set two properties:

- ▶ `ibm.dg.trc.print`: Set this property to 'mt' to invoke method trace.
- ▶ `ibm.dg.trc.methods`: Set this property to decide what to trace.

Use the `methods` parameter to control what is traced. To trace everything, set it to `methods=*. *`, which we do not recommend because you are certain to get overwhelmed by the amount of output. The `methods` parameter is formally defined as:

```
ibm.dg.trc.methods=[[!]method_spec[,...]]
```

Where `method_spec` is formally defined as:

```
{*|[*]classname[*]}.{*|[*]methodname[*]}[()]
```

The delimiter between the parts of the package name is a forward slash (/). The exclamation point (!) in the `methods` parameter is *not* an operator that allows you to tell the JVM not to trace the specified method or methods. The parentheses () that are in the `method_spec` define whether to trace method parameters.

The formatted trace output is written to stderr.

We traced the method calls in the `HelloCICSWorld` sample program by setting the properties `ibm.dg.trc.print=mt` and `ibm.dg.trc.methods=examples/HelloWorld/HelloCICSWorld.*()`. Example 8-9 on page 208 shows the information written to stderr. Message `JVMDG200`

shows the diagnostics properties set. The HelloCICSWorld class only has a main method, so the output shows entry and exit to main.

Example 8-9 Formatted method trace in stderr

```
JVMDG200: Diagnostics system property ibm.dg.trc.print=mt
JVMDG200: Diagnostics system property ibm.dg.trc.methods=examples/HelloWorld/HelloCICSWorld.*()
JVMDG200: Diagnostics system property ibm.jvm.events.output=event.log
10:11:59.353*0x349A8280 40011 > examples/HelloWorld/HelloCICSWorld.main Bytecode static method,
Signature: (Lcom/ibm/cics/server/CommAreaHolder;)
10:11:59.355 0x349A8280 40022 < examples/HelloWorld/HelloCICSWorld.main Bytecode static method
```

Writing trace data to stderr in real time is fine for low volume and non-performance-critical instances. For larger volumes and performance-critical tracing, you must not write formatted output to stderr. You can direct unformatted trace output to in-storage buffers or to one or more external files using buffered I/O.

Writing trace to buffers

The use of in-storage buffers for trace is an efficient method of running trace because no explicit I/O is performed until either a problem is detected or an API is used to snap the buffers to a file. To examine the trace data, you must snap or dump, and then format the buffers. Buffers are snapped when:

- ▶ An uncaught Java exception occurs.
- ▶ An operating system signal or exception occurs.
- ▶ The `com.ibm.jvm.Trace.snap()` Java API is called.
- ▶ The JVMRI `TraceSnap` function is called.

The resulting snap file is placed in the current working directory with a name in the format `Snapnnnn.yyyymmdd.hhmmss.th.process.trc`, where `nnnn` is a sequence number starting at 0001 (at JVM startup), `yyymmdd` is the current date, `hhmmss.th` is the current time, and `process` is the process identifier.

Use the `ibm.dg.trc.buffers` system property to specify the size of the buffers that are allocated for each thread that makes trace entries. See *SDK Diagnostics Guide*, SC34-6650 for more information.

Writing trace to an external file

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, which allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the `nodynamic` option of the `ibm.dg.trc.buffers` system property. For long-running trace runs, a `wrap` option is available to limit the file to a given size. See the `ibm.dg.trc.output` property in *SDK Diagnostics Guide*, SC34-6650, for details. You must use the trace formatter to format trace data from the file.

You can use the `-Dibm.dg.trc.external=mt` option in the properties file to send the method trace to the CICS auxiliary trace facility.

Note: Because of the buffering of trace data, if the normal JVM termination is not performed, residual trace buffers might not be flushed to the file. Snap dumps do not occur, and the trace bytes are not flushed except when a fatal operating-system signal is received. The buffers can, however, be extracted from a system dump if that is available.

Formatting trace

The trace formatter is a Java program that runs on any platform and can format a trace file from any platform. The formatter, which is shipped with the SDK in `core.jar`, also requires a file called `TraceFormat.dat`, which contains the formatting templates. This file is shipped in `jre/lib`.

You invoke the trace formatter by typing:

```
java com.ibm.jvm.format.TraceFormat input_filespec [output_filespec] [options]
```

Where `com.ibm.jvm.format.TraceFormat` is the traceformatter class, `input_filespec` is the name of the binary trace file to be formatted, and `output_filespec` is the optional output file name. If it is not specified, the default output file name is `input_filespec.fmt`.

CICS SJ domain tracing for JVMs

In addition to the trace points that the JVMs produce, CICS provides some standard trace points in the SJ (JVM) domain to trace the actions that CICS takes in setting up and managing JVMs and the shared class cache. These are available at CICS trace levels 0, 1 and 2.

You can activate the SJ domain trace points at levels 0, 1, and 2 using the CETR Component Trace screens. Select tracing by component, in the CICS Problem Determination Guide, to learn how to do this.

The SJ domain includes a level 2 trace point SJ 0224, which shows you a history of the programs that used each JVM.

“JVM domain trace points”, in the CICS Trace Entries manual, has details of all of the standard trace points in the SJ domain.

8.3.6 JVM class loader tracing

The z/OS Persistent Reusable JVM that CICS uses has three classpaths on which classes can be included:

- ▶ Trusted middleware class path
- ▶ Shareable application class path
- ▶ Standard class path

There is a class loader for each of these class paths. The status of a class within the JVM is dictated by its class loader. If a class is loaded by the wrong class loader, the behavior of the application can change, for example, a JVM can become unresettable.

There can be a large number of class and jar files on the classpaths, and you might not be able to easily identify on which classpath a class is included. Using JVM class loader tracing you can obtain which class loader is responsible for loading a class.

You enable class loader tracing by specifying the system property `ibm.verbose.cl` in the properties file for the JVM. The output is written to `stdout`.

You can trace loading for just one class or for many classes. You can use wildcard characters in the class name, for example:

- ▶ `ibm.cl.verbose=examples.HelloWorld.HelloCICSWorld`
- ▶ `ibm.cl.verbose=examples.HelloWorld.HelloCICSWorld,com.ibm.cics.server.Task`
- ▶ `ibm.cl.verbose=com.ibm.cics.server.T*`

Fully-qualified class name: The fully qualified class name must be specified.

We trace the loading of the sample CICS class `HelloCICSWorld` by adding `ibm.cl.verbose=examples.HelloWorld.HelloCICSWorld` to the systems properties file for the JVM. Example 8-10 shows the output. The output shows that `HelloCICSWorld` is loaded by the standard Java class loader. You can also obtain in which jar file or directory the class was found.

Example 8-10 JVM class loader trace output

```
ExtClassLoader attempting to find examples.HelloWorld.HelloCICSWorld
ExtClassLoader using classpath /usr/lpp/java142s/J1.4/lib/ext/dumpfmt.jar:/usr/lpp/java142s/J1.4/lib/ext/gskikm.jar:/usr/lpp/java14r
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/dumpfmt.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/gskikm.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ibmjcefpis.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ibmjceprovider.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ibmjce4758.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ibmjseprovider2.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ibmpkcs11impl.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/indicim.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/jaccess.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/ldapsec.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/oldcertpath.jar
ExtClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/lib/ext/recordio.jar
ExtClassLoader could not find examples.HelloWorld.HelloCICSWorld

MiddlewareClassLoader attempting to find examples.HelloWorld.HelloCICSWorld
MiddlewareClassLoader using classpath /SYSTEM/cicsts/cics630/lib/dfjcicras.jar:/SYSTEM/cicsts/cics630/lib/ras.jar:/SYSTEM/cicsts/cr
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcicras.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/ras.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjwrap.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjorjb.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcont.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcsi.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcics.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcdmn.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjfts.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjadjr.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/omgcos.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjname.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/websphere.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjcci.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/ctg/ctgclient.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/ctg/ctgserver.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/ctg/ccf.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /SYSTEM/cicsts/cics630/lib/dfjejbdd.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/standard/ejb/2_0/ejb20.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/standard/jta/1_0_1/jta-specr
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/java142s/J1.4/standard/jca/connector.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/db2710/db2710/jcc/classes/db2jcc.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/db2710/db2710/jcc/classes/db2jcc_javax.jar
MiddlewareClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /usr/lpp/db2710/db2710/jcc/classes/db2jcc_license_r
MiddlewareClassLoader could not find examples.HelloWorld.HelloCICSWorld

AppClassLoader attempting to find examples.HelloWorld.HelloCICSWorld
AppClassLoader using classpath /u/plowman/cics630/myprogs/myCICSPrograms.jar:/u/plowman/cics630/samples/dfjcics:/u/plowman/cics630/s
AppClassLoader could not find examples/HelloWorld/HelloCICSWorld.class in /u/plowman/cics630/myprogs/myCICSPrograms.jar
AppClassLoader found examples/HelloWorld/HelloCICSWorld.class in /u/plowman/cics630/samples/dfjcics
AppClassLoader found examples.HelloWorld.HelloCICSWorld
```

8.3.7 Shared classes diagnostics

Class sharing in the IBM Version 5.0 SDK offers a transparent and dynamic means of sharing all loaded classes, both application classes and system classes, and placing no restrictions on JVMs that are sharing the class data (unless runtime bytecode modification is used).

Sharing all immutable class data for an application between multiple JVMs has obvious benefits:

- ▶ The virtual memory footprint reduction when using more than one JVM instance can be significant.
- ▶ Loading classes from a populated cache is faster than loading classes from disk because the classes are already in memory and are already partially verified. Therefore, class sharing also benefits applications that regularly start new JVM instances doing similar tasks. The cost to populate an empty cache with a single JVM is minimal, and when more than one JVM is populating the cache concurrently, this activity is typically faster than both JVMs loading the classes from disk.

When you are running in shared classes mode, a number of diagnostics tools can help you. The verbose options are used at runtime to show cache activity. You can use the `printStats` and `printAllStats` utilities to analyze the contents of a shared class cache.

Verbose output

The verbose suboption of `-Xshareclasses` gives the most concise and simple diagnostic output on cache usage. Verbose output typically looks like Example 8-11.

Example 8-11 Sample output of `-Xshareclasses:name=myCache,verbose`

```
java -Xshareclasses:name=myCache,verbose -Xscmx10k HelloWorld
[-Xshareclasses verbose output enabled]
JVMSHRC158I Successfully created shared class cache "myCache"
JVMSHRC166I Attached to cache "myCache", size=10200 bytes
JVMSHRC096I WARNING: Shared Cache "myCache" is full. Use -Xscmx to set cache size.
Hello
JVMSHRC168I Total shared class bytes read=0. Total bytes stored=9284
```

The output in Example 8-11 shows that a new cache, called `myCache`, was created, which was only 10 kilobytes in size, and the cache filled up almost immediately. The message displayed on shutdown shows how many bytes were read or stored in the cache.

VerboseIO output

The `verboseIO` output is far more detailed and is used at runtime to show classes that are stored and found in the cache. You enable `verboseIO` output by using the `verboseIO` suboption of `-Xshareclasses`.

`VerboseIO` output provides information about the I/O activity occurring with the cache, with basic information about find and store calls. With a cold cache, you see trace like Example 8-12.

Example 8-12 Trace

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id
0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id
3... Failed.
```

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Failed.  
Storing class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

Each classloader is given a unique ID, and the bootstrap loader is always 0. In the trace in Example 8-12 on page 211, you see classloader 17 obeying the classloader hierarchy of asking its parents for the class. So each of its parents consequently asks the shared cache for the class. Because it does not yet exist in the cache, all the find calls fail and classloader 17 stores it.

After the class is stored, you see the output in Example 8-13.

Example 8-13 Trace output after the class is stored

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.  
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.  
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17...Succeeded.
```

Again, the classloader obeys the hierarchy, with its parents asking the cache for the class first. It succeeds for the correct classloader.

Verbose Helper, printStats utility, and printAllStats utility

There are other utilities that can give you more information about the class cache. You can enable them by using the following parameter in the JVM profile.

- ▶ -Xshareclasses:verboseHelper
- ▶ -Xshareclasses:printStats,name=<cache_name>
- ▶ -Xshareclasses:printAllStats,name=<cache_name>

More more information about the shared classes diagnose utilities is available in *SDK Diagnostics Guide*, SC34-6650.

8.4 Interactive debugging

In this section, we discuss several techniques that can perform an interactive debugging of the Java program running in CICS.

8.4.1 Execution diagnostic facility

You can use the execution diagnostic facility (EDF) to test an application program online without modifying the program or the program-preparation procedure. The CICS execution diagnostic facility is supported by the CICS-supplied transaction, CEDF.

Invoking CEDF: You can also invoke CEDF indirectly through another CICS-supplied transaction, CEDX, which enables you to specify the name of the transaction you want to debug. When this section refers to the CEDF transaction (for example, when it explains about CICS starting a new CEDF task) remember that the CEDX command might have invoked it.

EDF intercepts the execution of CICS commands in the application program at various points, which allows you to see what is happening. Each command is displayed before execution, and most are displayed after execution is complete. Screens that the application program sends are preserved, so you can converse with the application program during testing, just as a user does on a production system.

CICS Application Programming Guide, SC34-6231 has more information about EDF and user instructions for EDF.

When used with a JCICS program, CEDF intercepts the CICS commands that the Java Native Interface (JNI) programs invoke, which provide the interface between JCICS classes and CICS. There might not be an obvious relationship between the JCICS class method and the CICS command, for example, the HelloCICSWorld sample uses the `Task.out.println` method to send a message to the user terminal. The JNI program invokes the `SEND` command to write to the panel. Example 8-14 shows the line of Java source code to write to the panel (`t` is an instance of `Task`) and the image from CEDF showing the intercepted `SEND` command.

Example 8-14 Writing to the panel using `Task.out.println` method

```
Task.getTask().out.println("Hello from a Java CICS application");
```

```
TRANSACTION: JHE2 PROGRAM: DFJ$JHE2 TASK: 0006530 APPLID: SCSCPJA7 DISPLAY: 00
STATUS: ABOUT TO EXECUTE COMMAND
EXEC CICS SEND
  FROM ('.A&Hello from a Java CICS application..')
  LENGTH (39)
  NOHANDLE
```

8.4.2 Debugging using Rational Developer for System z

The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism that is provided in the Java 2 Platform. This architecture provides a set of APIs that allow the attachment of a remote debugger to a JVM. A number of third-party debug tools, including Rational Developer for System z, are available to exploit JPDA and can be used to attach to and debug a JVM that is running a Java program. Typically, the debug tool provides a graphical user interface that runs on a workstation and allows you to follow the application flow, set breakpoints and step through the application source code, and examine the values of variables.

When you start the JVM in debug mode, the JVMDI interface is activated and additional threads are started in the JVM. One of these threads handles communication with the remote debugger. The other threads monitor the application that is running in the JVM. You can issue commands in the remote debugger, for example, to set break points or to examine the values of variables in the application. These commands are activated by the listener and event handler threads in the JVM.

We use the debugger that is provided in Rational Developer for System z to show how to attach a debugger to a CICS JVM. We chose the HelloWorld sample program HelloCICSWorld in the getting started chapter as the application to be debugged.

JVM profile changes

To run a JVM in debug mode and allow a JPDA remote debugger to be attached, you must set some options in the JVM profile for the JVM:

- ▶ `-Xdebug=YES`

This is needed to start the JVM in debug mode (that is, with the JPDA interfaces active).

- ▶ `-Xrunjdwp=(suboption=...,suboption=...)`

This option specifies the details of the connection between the debugger and the CICS JVM. These details include the TCP/IP address to be used for the connection and the sequence in which the connection occurs. Different debuggers have different connection requirements and capabilities. Refer to the documentation that is provided with the debugger.

We specify the following suboptions for the connection between IBM Rational Software Development Platform and the CICS JVM:

```
Xrunjdwp=(transport=dt_socket,server=y,address=8000)
```

This set of suboptions specifies that:

- The standard TCP/IP socket connection mechanism is used.
- The JVM starts first (`server=y`) and waits for the debugger to attach to it.
- The CICS JVM listens on TCP/IP port 9876 for a debugger to attach to it.

- ▶ `REUSE=NO`

A JVM that was run in debug mode is not a candidate for reuse. Set this option to NO to ensure that the JVM is discarded after the debug session.

When you set these options in a JVM profile, any CICS JVM program that uses that profile runs in debug mode (and waits for attach from, or attempts to attach to a debugger). You must therefore ensure that the JVM profile applies only to programs that you want to debug.

Instead of configuring any of the CICS-supplied sample profiles for debug, you must create a separate JVM profile specifically for debug use, and set the appropriate CICS PROGRAM resource definition or use the CADP transaction to use this debug JVM profile.

We recommend that you copy the existing JVM profile to a new file and add the options to enable remote debugging. Use CADP to enable the transaction to be debugged by switching to the debug profile, which ensures that the JVM options (which influences the behavior of the Java application) are identical.

Rational Developer for System z configuration

We use the debugger in Rational Developer for System z. The source code for the application you want to debug must be loaded into the Rational Developer for System z. We use the Java project that we created in Chapter 5., “Writing Java 5 applications for CICS” on page 75.

Adding dfjcics.jar: You must add dfjcics.jar to the Java build path for your project because this application uses JCICS.

Figure 8-3 on page 215 shows the Java perspective within Rational Developer for System z displaying the JCICS HelloWorld project:

- ▶ The package explorer pane on the left shows the project, package, and class hierarchy. It also shows dfjcics.jar on the build path.
- ▶ The center pane shows the source code of HelloWorld. We set a breakpoint at line 15 of the source, which is indicated by the blue button to the left of the pane.

- ▶ Hovering over the button displays the text shown.
- ▶ Double clicking in the left edge of the pane toggles a line breakpoint on and off.

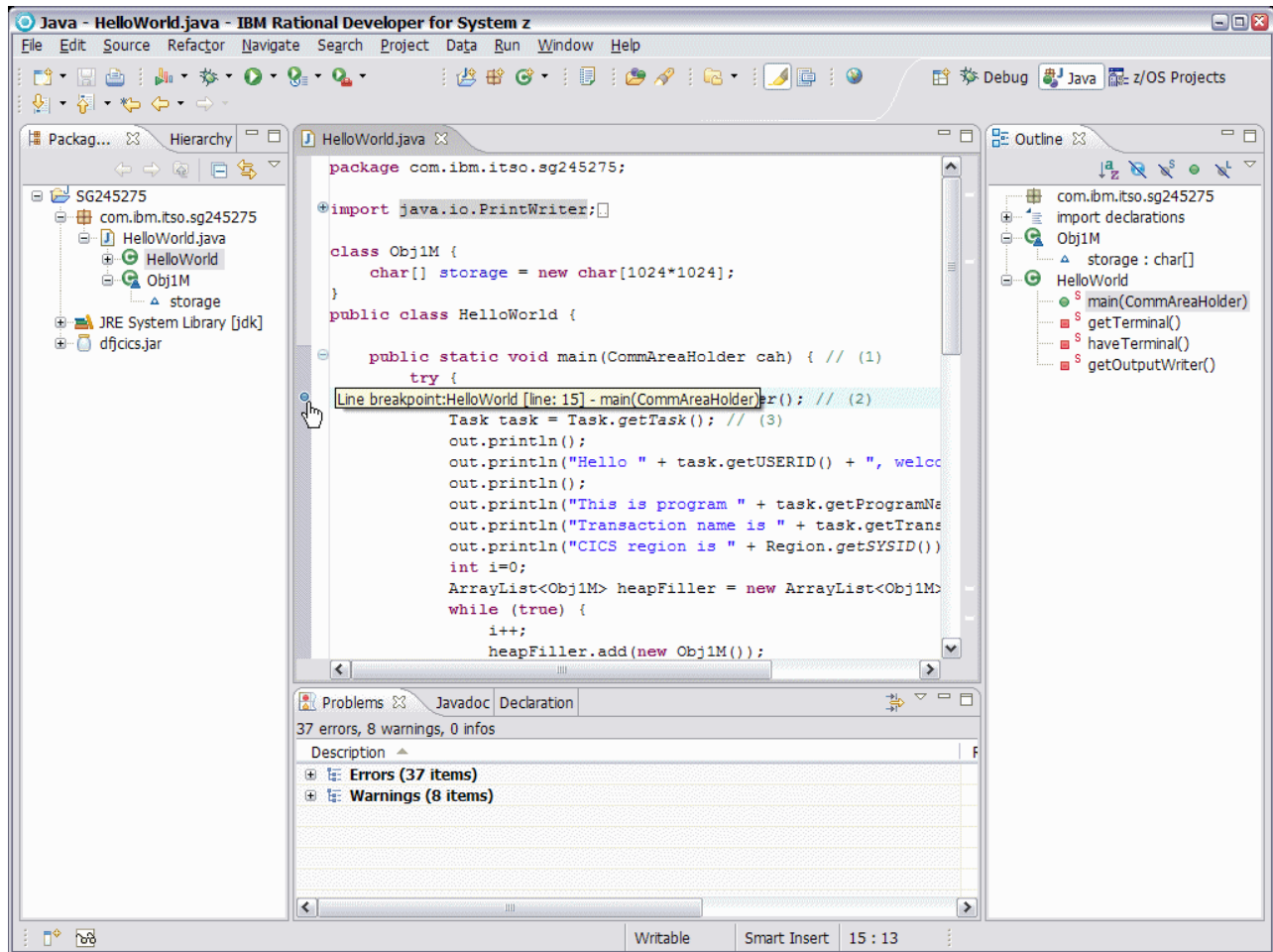




Figure 8-3 Sample HelloWorld project

You must create a Remote Java Application configuration within the Debug Perspective that specifies:

- ▶ The IP address (or host name) of the z/OS system that hosts the CICS region.
- ▶ The TCP/IP port number that the CICS JVM is using. (This is the same number that is specified to CICS on the Xrunjdw option in the JVM profile.)
- ▶ That a standard TCP/IP socket connection (Socket Attach) is to be used.

To open the Debug configuration window:

1. Click the arrow to the right of the  icon in the toolbar.
2. Click the  icon in the menu. Figure 8-4 on page 216 is displayed.

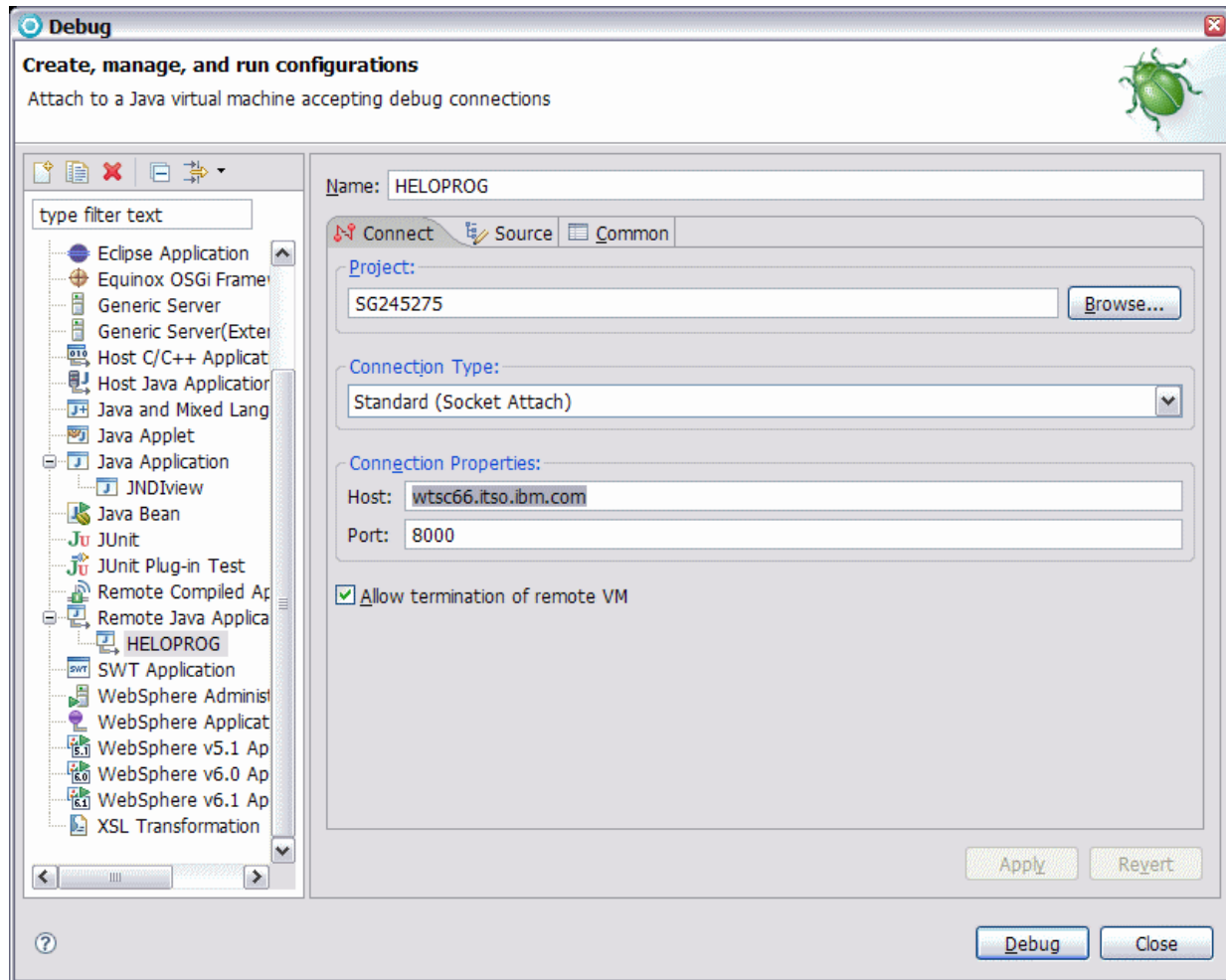


Figure 8-4 Configuring remote Java application

Debugging the application

You are now ready to start the interactive debugging session:

1. Enter the transaction identifier on CICS to run the Java program to be debugged. For our example, we enter HELO, which starts the JVM in debug mode. Your terminal session will hang. The JVM is waiting for the connection from the remote debugger.
2. Start the debug configuration previously created in the Rational Developer for System z. The debugger connects to the JVM in CICS, the debug perspective opens, and you can start debugging. Figure 8-5 on page 217 shows the debug perspective.
3. From the debug perspective you can step through the code of your class, inspect and change variables, and set new breakpoints. If the source code is available, you can also interactively debug other classes that were invoked during the execution of your application program.

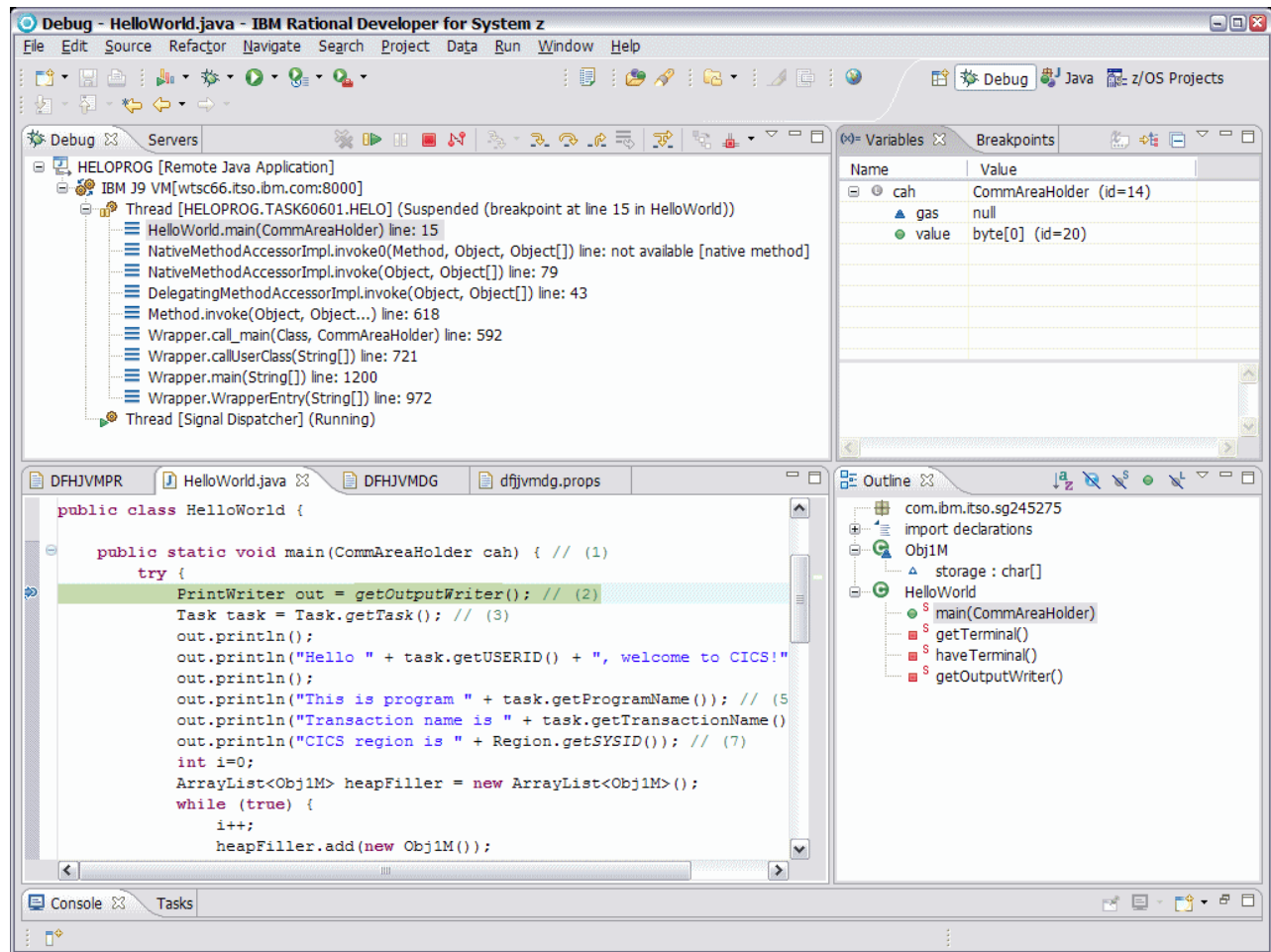


Figure 8-5 Debug perspective in Rational Developer for System z

Syncing the byte code and source code: Be careful to keep the byte code and the source code in sync, that is, make sure that you re-export the byte code to the host after you change the source code on the development workstation.

8.4.3 CICS Application Debugging Profile

Using the CICS Application Debugging Profile (CADP) CICS-supplied transaction you can manage application debugging profiles for both traditional and Java programs.

The benefit of using CADP is that you can dynamically switch the JVM profile at invocation time based on certain runtime characteristics. As we mentioned in previous section, if you want to debug a Java program in CICS, you need a special JVMProfile that specifies both REUSE=NO and -Xdebug=YES and the JDWP options, such as -Xrunjdwp:transport=dt_socket,address=workstationHostname:debuggingPortNumber. Each developer typically needs their own variant of this debugging profile.

Look at this example: If you want to debug a particular Java program, you can tell CADP that whenever user ID 'FRED' runs Java program 'com.ibm.cics.example.MyExample', CICS dynamically switches the JVMProfile from the normal one (typically DFHJVMPJR) to a custom one, such as FREDDEBG, which has Fred's personal debugging parameters. Other users can set up similar profiles in the same CICS region.

At runtime, CICS spots that MyExample is to be linked if the current USERID is FRED. The debugging JVM profile is used rather than DFHJVMPR, which in turn results in the JVM attaching to the debugger that is running on the workstation. It is a flexible solution that works nicely with multiple users. It also integrates reasonably well with the CADP-based debugging for other languages, such as COBOL and PL/I. Figure 8-6 shows the CICS Application Debugging Profile Manager.

CADP also has a Web-based interface in CICS so that the developer can set up the CADP debugging profiles without ever using a 3270 terminal. For more information, check the CADP chapter in the CICS manual, *CICS Supplied Transactions*, SC34-6817.

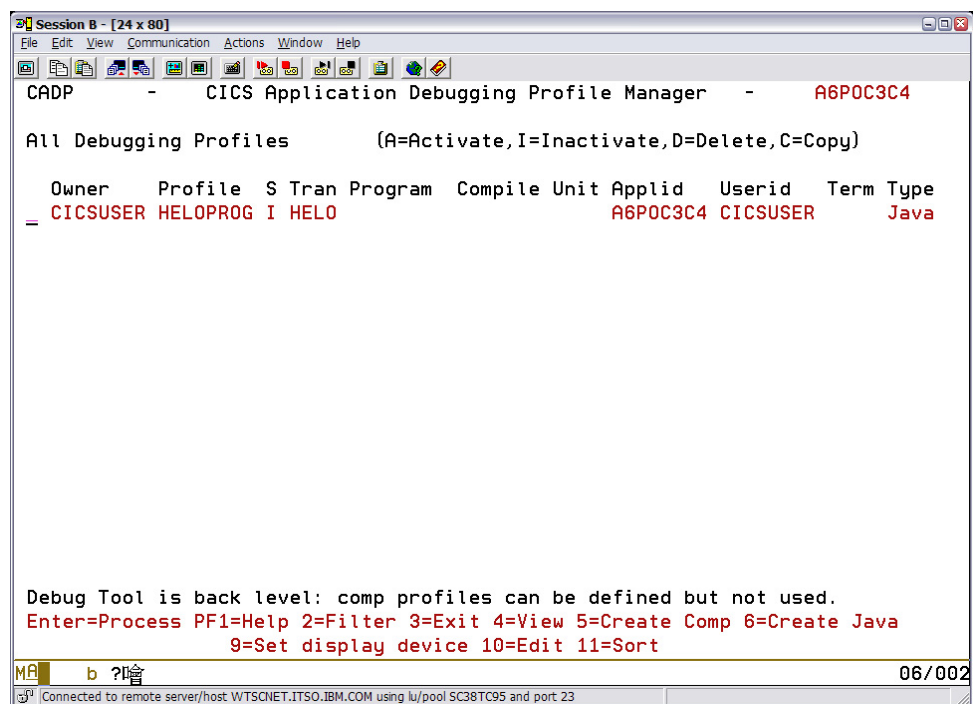


Figure 8-6 CICS Application Debugging Profile Manager

8.4.4 The CICS JVM plug-in mechanism

In addition to the standard JPDA debug interfaces in the JVM, CICS provides a set of interception points (or plug-ins) in the CICS Java middleware, which can be of value to developers for debugging applications. These interception points allow additional Java programs to be inserted immediately before and after the application Java code is run. Information about the application (for example, classname and method name) is made available to the plug-in programs. The plug-in programs can also use the JCICS API to obtain information about the application. You can use these interception points in conjunction with the standard JPDA interfaces to provide additional CICS-specific debug facilities. You can also use them for purposes other than debugging, in a similar way to user exit points in CICS.

The programming interface consists of two Java interfaces:

- DebugControl** Defines the method calls that can be made to a user-supplied implementation
- Plugin** Provides a general purpose interface for registering the plug-in implementation

These interfaces are supplied in dfjwrap.jar and documented in JAVADOC HTML.

These plug-ins are not described further in this book. Refer to *Java Applications in CICS*, SC34-6238, for more information.



Performance for Java in CICS Transaction Server Version 3

In this chapter, we first look at the Continuous Java virtual machine (JVM), and we describe it in detail. We look at the potential problems and the things that you must avoid that might impact performance. We recommend that you review Chapter 16, “Java applications using a Java virtual machine (JVM): improving performance,” of the *CICS Performance Guide*, SC34-6833-01, for good performance tips.

9.1 Reusable Java virtual machine

In this section, we discuss reusable JVM.

9.1.1 CICS Task Control Blocks and the Java virtual machine

CICS has a small number of Task Control Blocks (TCBs), which it uses when running application programs. Of these, the Quasi Reentrant (QR) TCB is used to single thread the entire workload. At any point in time there can be a number of transactions running in a CICS region. Only one of these has use of the QR TCB at any time. If that application needs to pause for some reason, for example, when writing to a data set or waiting for a resource lock, CICS suspends it, which allows another applications to gain control of the QR TCB and carry out some work; therefore, CICS can serialize the execution of all the applications that it has concurrently running, which prevents the actions of one from interfering with those of any others.

This mechanism was developed for applications that are written in fully compiled languages, such as COBOL, PL/I, or C. However, the introduction of support for Java programs presented some additional challenges, which CICS has had to address.

Pure Java requires an execution environment provided by a JVM that is written for the operating system on which it runs. On z/OS, a JVM can sometimes issue blocking calls, such as MVS waits, which cause the TCB that is used by the JVM to go into a wait state. If a JVM was started on the QR TCB, then blocking calls causes the entire CICS workload to pause. To address this issue, CICS Transaction Server Version 1.3 introduced a new type of TCB known as the J8 TCB. CICS uses one of these for each JVM that it starts. At the same time, from CICS Transaction Server Version 2.3 there is a J9 TCB to be added. And CICS Transaction Server Version 3.2 keeps J8 TCB and J9 TCB to support Java workloads.

The JVM that CICS uses supports multithreaded applications. However, only one application can run in a JVM at any one time. More than one TCB can be used when a single CICS region must support multiple concurrent Java applications. In this way, a CICS region can have a single pool of J8 and J9 TCBs to support its Java workloads.

All new CICS transactions start on the QR TCB. If the transaction makes use of a Java program, then CICS switches to either a J8 or a J9 TCB and runs the program under the control of a JVM there. When the program terminates, or if it needs to access a CICS-managed resource, then CICS switches control back to the QR TCB for that piece of processing. In this way, the JVM can pause without interrupting the rest of the CICS workload, so serialization of access to CICS resources and the management of the start and end of all transactions occurs using the QR TCB.

9.1.2 The reusable Java virtual machine

There are significant CPU costs in starting and stopping a JVM, which are often greater than the costs that are incurred from running of an application within a JVM. If a JVM can be serially reused by a number of CICS programs, the overhead of starting and stopping the JVM is confined to the first and last program in the sequence and those in between gain significant performance benefits from this.

However, if a JVM is reused, the potential exists for one Java application program to leave around objects whose state can interfere with the successful execution of programs that later reuse the same JVM. EJBs are inherently self-contained and as such cannot leave state around after their execution. Individual Java applications, on the other hand, tend to treat the JVM where they run as something to which they have exclusive use. They might assume that

the JVM is destroyed when the application finishes, and that all object instances, created during program execution, get removed as a result of the termination of the JVM. This assumption is not valid when a JVM is serially reused and side effects, resulting from objects remaining in the JVM's storage heap following the termination of one application, might interfere with other applications that later reuse the same JVM.

To address this issue, the IBM Development Kit for z/OS, Java 2 Technology Edition (SDK) provides a JVM with a modified heap structure, and this JVM can be driven in an attempt to reset its storage areas between separate program invocations. You can configure CICS Transaction Server Version 2 to make use of this mechanism. Reset processing extends the operation of the JVM but does have some adverse effect on its performance characteristics.

Not all applications cause these problems and, for them, this reset processing is an unnecessary overhead. CICS Transaction Server Version 2.3 and the SDK 1.4.1 provide support for another JVM configuration, which offers the potential for reuse, but that places responsibility for reset processing onto the applications that run there. This provides significant performance benefits but does present opportunities for badly behaved programs to interfere with others that reuse the same JVM.

9.1.3 Removing resettable mode for JVMs in CICS Transaction Server 3.2

In CICS Transaction Server for z/OS, Version 3 Release 2, resettable JVMs, which were reset between each use, are no longer supported. Any Java programs that ran in resettable JVMs must be migrated to run in continuous JVMs. Resettable JVMs had the option `REUSE=RESET` in their JVM profiles (or the older option `Xresettable=YES`).

Although this process enforced serial isolation for programs running in the JVM, the time and CPU usage that is required for a JVM reset reduced the performance of a resettable JVM compared to the performance of a continuous JVM. Resettable JVMs were also incompatible with future versions of Java, whereas continuous JVMs are compatible with future versions of Java.

An application that is coded with attention to the state of the JVM and to the items in static storage can operate safely in a continuous JVM without the JVM reset. If you need to police the use of any APIs in the continuous JVM, you can use the Java security manager to do this.

The migration process for Java programs that run in a resettable JVM involves checking that the Java programs do not contain any code that might have an unwanted effect on serial isolation when the continuous JVM is reused by a subsequent program. The CICS JVM Application Isolation Utility, a code checking and reporting utility, is provided with CICS Transaction Server for z/OS, Version 3 Release 2 to help identify areas where you must check the behavior of Java programs that were designed to run in resettable JVMs.

Configuration and tuning for continuous JVMs is simpler than it was for resettable JVMs. Your choice of class path is more straightforward, and there are fewer storage settings to tune. When you migrate an application to run in a continuous JVM, you probably need to merge some of your existing storage settings. Your existing class path options are accepted for migration purposes, and CICS issues a warning message about those options that are obsolete. [Reference: *Java Applications in CICS Version 3 Release 2*, SC34-6825-01.]

9.2 Shared Class Cache facility

In this section, we discuss the Shared Class Cache facility. We begin by providing an overview of the Shared Class Cache and then we describe the benefits of using it.

9.2.1 Overview of the Shared Class Cache facility

The principle of sharing loaded classes between Java virtual machine (JVM) processes is not new. Actually the SDK V1.4 provided a mechanism that allows Java classes to be cached centrally and shared between different JVMs. Also, CICS Transaction Server Version 2.3 introduces a Shared Class Cache facility that extends this function to some, or all, of the JVMs that it controls.

Now CICS Transaction Server for z/OS, Version 3 Release 2 supports the JVM provided by the 31-bit version of IBM SDK for z/OS, Java 2 Technology Edition, Version 5, as an alternative to the JVM provided by Version 1.4.2 of the SDK. You can choose to migrate some or all of your CICS regions from Java 1.4.2 to Java 5 to benefit from the new Java language features, and also the improvements to execution technology in the IBM SDK for z/OS, V5, including improved garbage collection and simpler class sharing. [Reference: *Java Applications in CICS Version 3 Release 2*, SC34-6825-01.]

The new Shared Classes feature in the IBM implementation of version 5.0 of the Java platform offers a completely transparent and dynamic means of sharing all loaded classes that places no restrictions on the JVMs that share the class data. This feature offers a straightforward and flexible solution for reducing virtual memory footprint and improving startup time, and there are few applications that will not benefit from it. In this section, we explore how the feature works, how to use it, and when to use it, along with some of the features that it provides.

The feature in the IBM z/OS® 1.4.2 JVM used a master JVM to populate a class cache that was then shared by worker JVMs, but the *master/worker* mechanism is replaced in the 5.0 JVM. In the 5.0 JVM, no JVM owns the cache, and there is no master/subordinate JVM concept; instead, any number of JVMs can read and write to the cache concurrently. The IBM implementation of the 5.0 JVM takes the concept a step further by allowing all system and application classes to be stored in a persistent dynamic class cache in shared memory. This Shared Classes feature is supported on all of the platforms on which the IBM implementation of the JVM ships. Figure 9-1 shows a view of the Shared Class Cache.

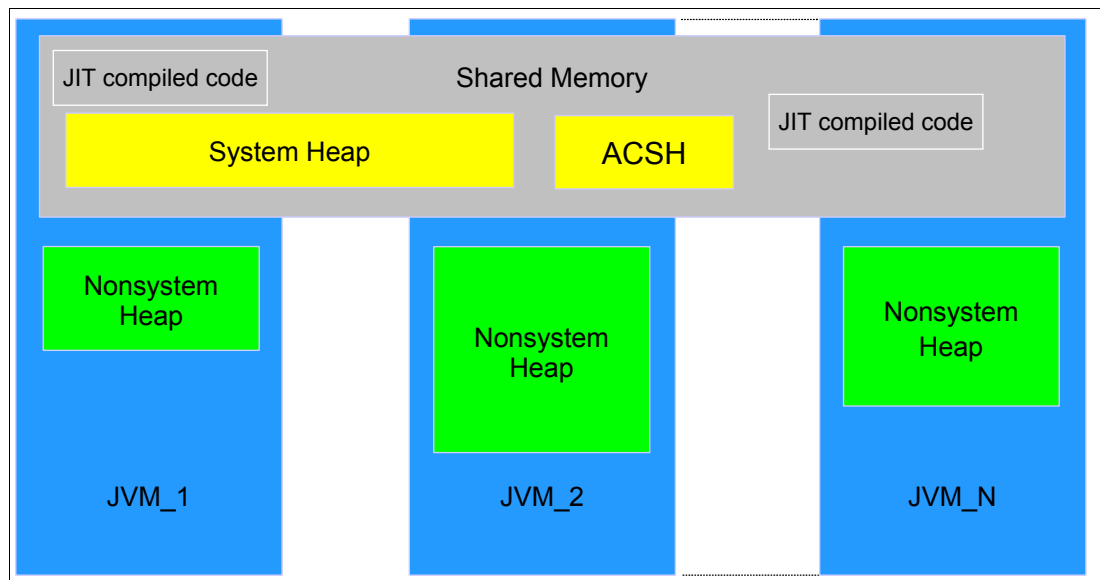


Figure 9-1 One view of the Shared Class Cache

Enabling class sharing

The Shared Classes feature was designed from the ground up to be an option that you can just switch on and forget about, yet it provides very powerful scope for reducing virtual memory footprint and improving JVM startup time. For this reason, it is best suited to environments where more than one JVM is running similar code or where a JVM is regularly restarted. [Reference: <http://www.ibm.com/developerworks/java/library/j-ibmjava4/#4>]

You enable class sharing by adding `-Xshareclasses[:name=<cachename>]` to an existing Java command line. When the JVM starts up, it looks for a class cache of the name given (if no name is provided, it chooses a default name), and it either connects to an existing cache or creates a new one, as required.

You specify cache size using the parameter `-Xscmx<size>[klm]`, which only applies if the JVM creates a new cache. If this option is omitted, a platform-dependent default value is chosen (typically 16 MB). Note that there are operating system settings that can limit the amount of shared memory to allocate. For z/OS, because the virtual address space of a process is shared between the shared class cache and the Java heap, increasing the maximum size of the Java heap reduces the size of the shared class cache that you can create.

Benefits of using the Shared Class Cache

The Shared Class Cache facility offers a number of benefits to clients. Java classes are loaded once per CICS region rather than once per JVM, which reduces the class loading overhead of each JVM startup in JVM ships. It also reduces the overall storage requirement for the JVM ships by storing a persistent dynamic class cache in shared memory, instead of one in each JVM's storage heap.

9.3 Things to avoid

In this section, we review potential problems and things you must avoid that can impact performance. These include:

- ▶ JVM stealing
JVM has to restart.
- ▶ Using application classpath
Use the Shareable classpath; otherwise, classes are physically reloaded each time.
- ▶ Excessive garbage collection
Make sure heaps are big enough for 101 trans when CICS does GC anyway.

9.3.1 Java virtual machine stealing

If a JVM is required to execute a Java program, CICS chooses to reuse a JVM that is not currently in use but initialized with the same JVM profile as the new request.

If no such JVM is available and the CICS region is not a MAXJVMTCB, CICS attaches a new J8 TCB and initializes a new JVM.

If no such JVM is available and the CICS region is at MAXJVMTCB, CICS steals a currently unused JVM, as shown in Figure 9-2 on page 226. Because this JVM is not using the same profile, it must be reinitialized.

If all of the MAXJVMTCB JVMs are being used, the new request must wait for a JVM to become available.

J8 TCBs that are used for Java do not come out of the MAXJVM pool. The JVM Pool is separate.

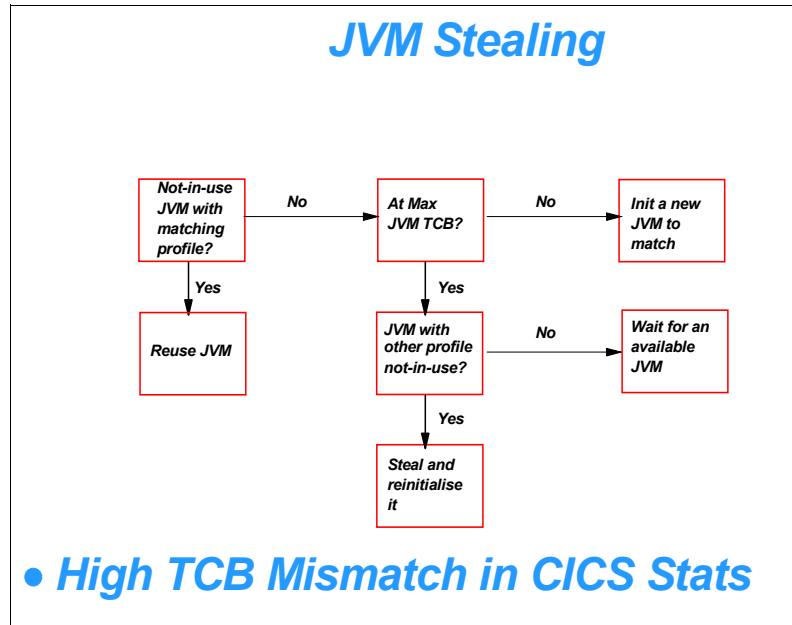


Figure 9-2 JVM stealing

9.3.2 Using application classpath

Class path is when we put our application programs on the Classpath and not the Shareable classpath. Figure 9-3 shows some classpath problems.

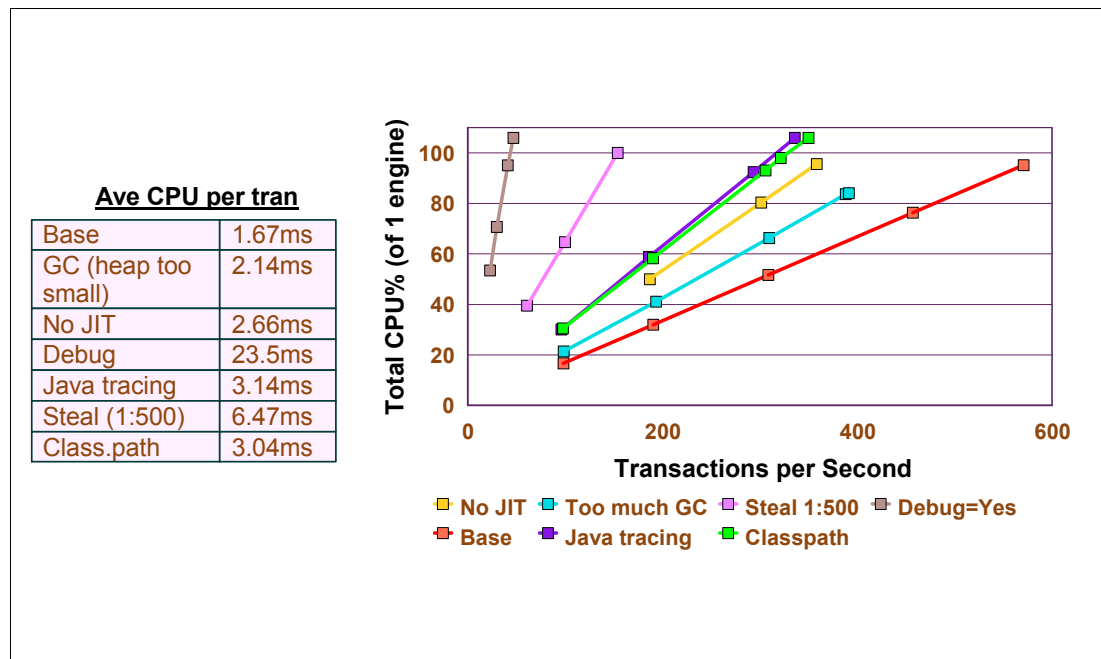


Figure 9-3 Classpath problems

9.3.3 Excessive garbage collection

In this section, we provide brief points about excessive garbage collection.

Excessive garbage collection:

- ▶ GC is where the heap is too small to support 101 transactions, so there is additional garbage collection going on.
- ▶ CICS forces GC every 101 transactions per JVM.
- ▶ When heaps are too small:
 - Additional GC is triggered.
 - This shows as a spike in JVMRTIME.
- ▶ Use VERBOSE=GC to investigate.
- ▶ More details on GC and heap sizing are provided later.
- ▶ The base is a normal, well-performing, small Java application.
- ▶ No JIT is where `java.compile=none` was specified.
- ▶ Debug is where `USE_LIBJVM_G=YES` was left in the JVM Profile.
- ▶ Java tracing is where we left `ibm.dg.trc.maximal=mt` in the property file and activated method level java tracing.
- ▶ Steal is where we force the JVM to be stolen so that another JVM with a different profile can use the TCB.
- ▶ Class path is when we put out application programs on the Classpath and not the Shareable classpath.

9.4 IBM zSeries Application Assist Processor specialty engines

In this section, we describe the new IBM zSeries Application Assist Processor (zAAP), which you can configure on the IBM z990, z890, z9, and z10 servers. The zAAP is an attractively priced specialized processing unit that provides an economical Java execution environment for customers who want the traditional Qualities of Service and the integration advantages of the zSeries platform.

9.4.1 zAAP introduction

In the e-business on demand era, business requirements change more frequently than ever. To stay current and competitive, businesses are developing new strategic Web-based applications. Java adoption continues to accelerate as an open programming model, but these applications typically require more IT resources than traditional applications because levels of abstraction, code generation, and reuse. Unfortunately, IT budgets are not keeping pace with these needs, forcing customers to seek more cost effective and productive ways of to deploy new Java technology-based applications.

Using the zAAP optional assist feature you can purchase additional processing power exclusively for Java application execution without affecting the total MSU rating or machine model designation.

zAAPs are designed to operate asynchronously with the general CPs to execute Java programming under control of the IBM Java Virtual Machine (JVM). This is an important point because zAAPs can only help execute Java applications and application servers that use the

IBM JVM. You can execute the IBM JVM processing cycles on the configured zAAPs with no modifications to the Java applications.

9.4.2 zAAP benefits

Using zAAPs you can:

- ▶ Simplify and reduce server infrastructures by integrating e-business Java Web applications next to mission critical data for high performance, reliability, availability, and security.
- ▶ Maximize the value of your zSeries investment through increased system productivity, which you achieve by reducing the demands and capacity requirements on general purpose processors; therefore, making those processors available for reallocation to other zSeries workloads.
- ▶ Lower the overall cost of computing for WebSphere Application Server and other Java technology-based applications through hardware, software, and maintenance savings.

When configured with general purpose processors within logical partitions that are running z/OS (or z/OS.e), zAAPs can help increase general purpose processor productivity and can contribute to lowering the overall cost of computing for z/OS Java technology-based applications.

The amount of general purpose processor savings vary based on the amount of Java application code that is executed by one or more zAAPs. This is dependent on the amount of Java cycles that the relevant applications use and on the zAAP execution mode that you select.

Execution of the Java applications on zAAPs, within the same z/OS LPAR as their associated database subsystems, can also help simplify the server infrastructures and improve operational efficiencies, for example, use of zAAPs can reduce the number of TCP/IP programming stacks, firewalls, and physical interconnections (and their associated processing latencies) that might otherwise be required when the application servers and their database servers are deployed on separate physical server platforms.

9.4.3 zAAP requirements

Hardware requirements for the zAAP are the z990, z890, z9, or z10 servers. On z990, z890, z9, or z10 servers, PUs characterized as zAAPs within a configuration are grouped into the ICF/IFL/zAAP processor pool. The ICF/IFL/zAAP processor pool appears on the hardware console as ICF processors. The number of ICFs shown is the sum of IFL, ICF, and zAAP processors that are characterized on the server.

zAAPs are designed to operate asynchronously with the general CPs to execute Java programming under the control of the IBM Java Virtual Machine (JVM). The IBM JVM processing cycles can be executed on the configured zAAPs with no anticipated modifications to the Java applications.

To exploit a zAAP, the operating system must be migrated to the following levels of software:

- ▶ z/OS V1R6 (or z/OS.e V1R6)
- ▶ The IBM SDK for z/OS
- ▶ Java 2 Technology Edition V1.4 with a PTF for APAR PQ86689
- ▶ For WebSphere-based Java workloads, WebSphere Version 5.1 or above

9.4.4 zAAP workflow

When a z/OS logical partition is configured, both CPs and zAAPs are defined as necessary to support the planned Java and non-Java workloads. zAAPs can be configured as initially online or reserved for subsequent use by z/OS as necessary. Pay attention that zAAPs cannot be IPLed, and at least one central processor is required for each z/OS partition.

zAAPs are defined as either shared by other logical partitions or dedicated to a specific partition; however, both central processors and the zAAPs for each partition have the same shared or dedicated attribute. For a given partition, you cannot define shared central processors and dedicated zAAPs or dedicated central processors and shared zAAPs. PR/SM configures shared zAAPs from the same pool of shared special purpose processors as ICFs and IFLs. Collectively, all shared ICFs, IFLs, and zAAPs also dynamically share in the processing requirements for all three special purpose processor types as controlled by PR/SM.

Integrated Facility for Applications (IFA) eligible work can run on both IFAs and standard processors. There are options that specify how the Java execution cycles are dispatched. You can specify that standard processors do not run any IFA-eligible work unless there are no IFAs operational in the partition. You can also prevent IFA-eligible work from running on standard processors because of software licensing considerations.

Figure 9-4 on page 230 shows the execution of a Java unit of work, as follows:

- ▶ Initially the Java code is dispatched on a standard processor (CP) and any other unit of work.
- ▶ Before the Java code gets executed on a Java machine (JVM), JVM signals to the dispatcher that the current unit of work is zAAP-eligible work.
- ▶ When the current unit of work releases control, the dispatcher places it in the zAAP dispatcher work queue. When a zAAP processor becomes available the dispatcher selects the highest priority work from the zAAP work queue and dispatches it on the zAAP processor.
- ▶ A zAAP-eligible unit of work can be executed on a zAAP (if a zAAP is available). Work executing on the zAAP inherits the dispatching priority from the execution on the regular CP. The Java application code executes on this zAAP (also called an IFA, or Integrated Facility for Application) processor. The MVS dispatcher dispatches the Java code to the zAAP processor unit.
- ▶ When the Java machine finishes processing (the unit of work finishes executing the Java code,) it signals to the dispatcher that the current unit of work is not eligible for zAAP processing anymore. When the unit of work releases control it is placed in the dispatcher "standard logical processor" work queue.
- ▶ The JVM returns to the WebSphere code that is running on the standard processor CP.

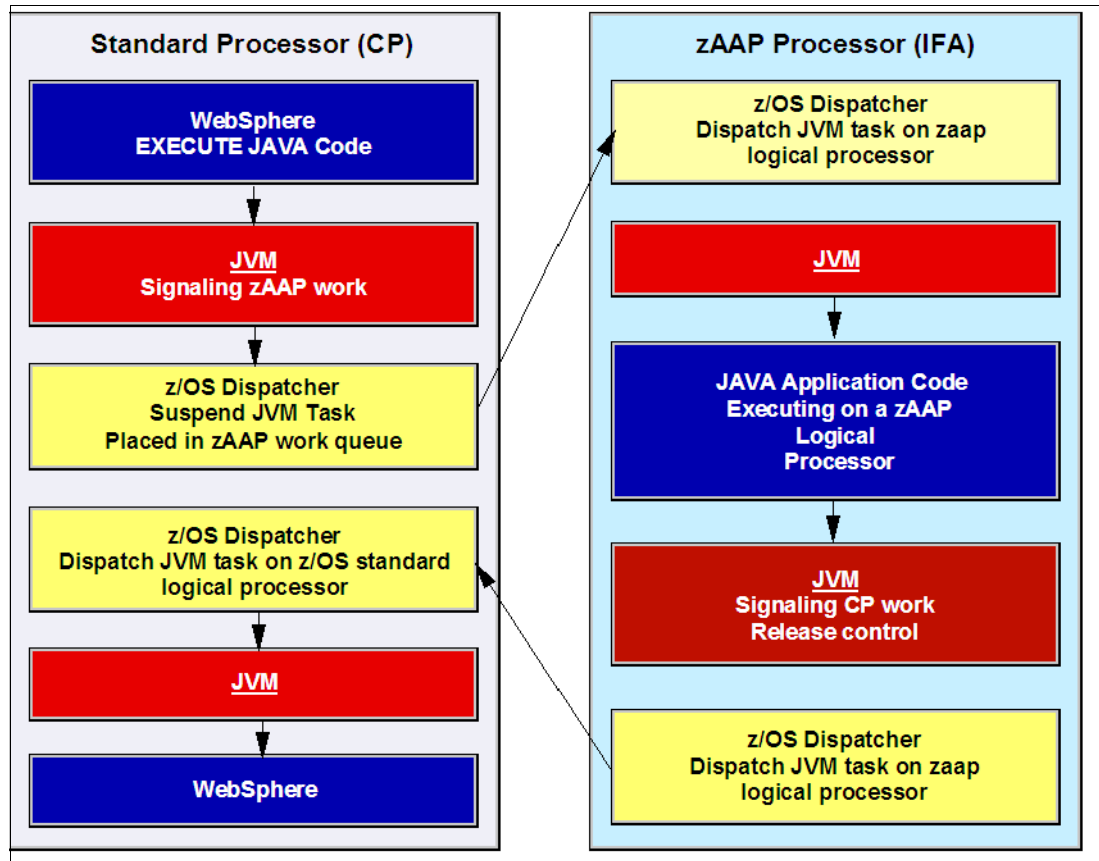


Figure 9-4 zAAP workflow of a Java unit of work

9.4.5 Using zAAPs in JVM

SDK1.4.1 has specific JVM options to handle zAAPs. However, if you are on z/OS V1R6 (or higher version) and want to enable zAAPs on your server, use the defaults. No additional setup is needed.

The JVM options for zAAP processing are:

► -Xifa:on

Enables Java work to run on a zAAP if any zAAPs are available. This is the default. Only code written in Java and Java system native code is enabled to run on a zAAP. This design point is achieved in the Java support by requesting a switch to a zAAP for qualifying work and a switch request from a zAAP to a general purpose processor when non-qualifying work is encountered. This option is honored only with z/OS V1R6.

► -Xifa:off


Disables the use of zAAPs.

- -Xifa:projectn

Designed to estimate projected zAAP usage and write this information to STDOUT at intervals of n minutes. A value of 0 indicates that information is only written when Java terminates. The interval requested is not honored exactly. Messages are written after a switch is encountered for work that is considered eligible for off load to a zAAP or returning from that state. As a result, messages can be delayed during idle periods.

- -Xifa:force

Designed to force Java to continue attempting to use zAAPs, even if none are available. This is typically specified for the purpose of collecting RMF/SMF data to assess potential zAAP use.



Performance tools for Java in CICS Transaction Server Version 3

In this chapter, we introduce some important JVM performance monitoring and analysis tools on z/OS. These include:

- ▶ CICS Explorer:
 - IBM CICS Explorer: New Face of CICS. Integration point for CICS tooling with rich CICS views, data, and methods.
- ▶ CICS PA:
 - CICS Performance Analyzer for z/OS is a reporting tool for analyzing and tuning the performance of your CICS systems.
- ▶ CICSplex System Management:
 - CICSplex SM (commonly known as CPSM) is part of CICS Transaction Server and using it you can manage multiple CICS systems from a single control point.
- ▶ OMEGAMON XE for CICS on z/OS:
 - OMEGAMON XE for CICS on z/OS provides the capability to monitor CICS Transaction Server (TS) environments.

10.1 CICS Explorer

The CICS Explorer is intended to act as the tooling integration point for the CICS runtime and to provide a rich set of CICS views, data, methods, and features, easily extensible by IBM, other vendors, and customers to deliver integrated solutions to key CICS users. The features offered by the CICS Explorer include:

- ▶ Common, intuitive, Eclipse-based environment for architects, developers, administrators, system programmers, and operators.
- ▶ Task-oriented views provide integrated access to broad range of data.
- ▶ Now packaged with Rational Developer for System z V7.5.
- ▶ Integration point for CICS TS, CICS Tools, Rational Tools, and others.
- ▶ Extensible by ISVs, SIs, and customers using the Software Development Kit bundled as part of the CICS Explorer.
- ▶ Dynamic resource relationships: See both uses and where used relationships for deployed CICS resources, for example, right-click a file to see all of the programs that use it (requires CICS IA licence).
- ▶ Point-and-click navigation: Much easier to follow a sequence of resource relationships.
- ▶ Data filtering: Helps to reduce the volume of data displayed, so that you can identify the required relationships more easily.
- ▶ Dynamic perspectives: Resize, reorder, or sort columns. Move, tab, or resize views.
- ▶ Powerful online help: Browse, search, and print documentation with context-sensitive help and text search capability. The help is displayed in a Help view in the workbench, in a separate Help Contents window, or in an external browser window.

The CICS Explorer is not a replacement for the CPSM WUI. The CICSplex SM WUI provides browser-based access to all CICSplex SM function, and will continue to do so. The CICS Explorer provides an installed client that provides a rich, integrated user experience.

The CICS Explorer does not support editing of CICS resources at this time.

It is not our intention in this section to discuss all of the features and advantages of using TS Explorer. For complete information, see the CICS TS Explorer documentation.

10.1.1 System requirements

Table 10-1 shows the system requirements for the CICS Explorer. The CICS Explorer also requires a CICSplex SM WUI server (CICS TS v3.1 or later) to connect to.

Table 10-1

Operating System	Software	Hardware
Linux, Windows	CICS Explorer runs under the control of, or in conjunction with, the following programs and their subsequent releases, or their equivalents: <ul style="list-style-type: none">▶ Eclipse V3.3 (included)▶ Windows: Release to be determined▶ Linux: Release to be determined	CICS Explorer runs on any hardware configuration that is supported by the licensed programs specified below.

Operating System	Software	Hardware
z/OS	Required licensed programs: <ul style="list-style-type: none"> ▶ IBM z/OS, Version 1.7 or later ▶ CICS Transaction Server for z/OS V3.1 V3.2 (5655-M15) 	CICS Explorer runs on any hardware configuration supported by the licensed programs specified below.

TS Explorer scene layout

A connected TS Explorer view looks similar to Figure 10-1. The window layout can vary a little, depending on what you are doing in the Explorer, but you can typically view it as three panels: left, top right, bottom right, and a status bar and the menus.

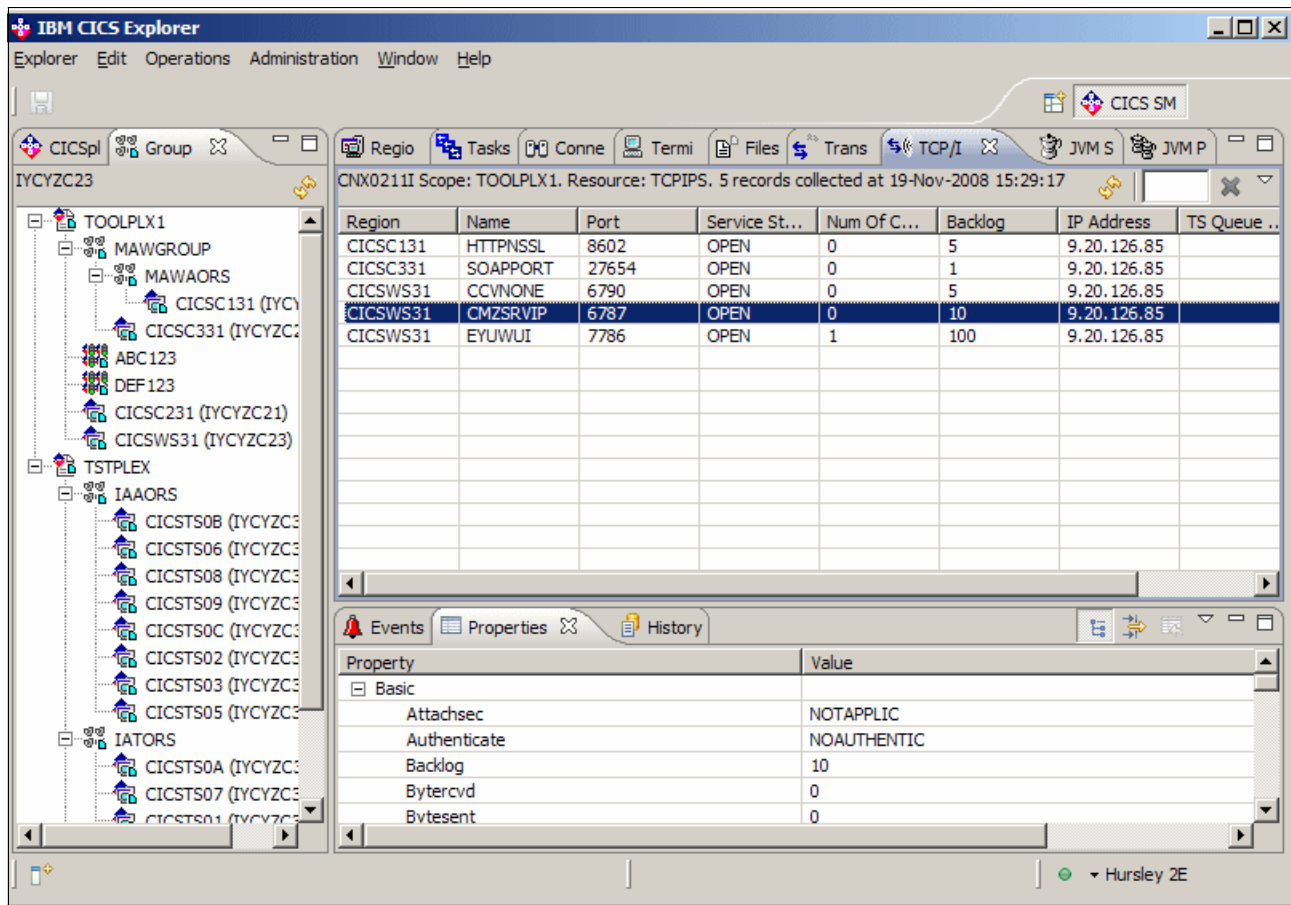


Figure 10-1 Typical Explorer view

The left panel shows the CPSM CICSplexes and the CICS regions. They contain what we can view through the server that we are connected to. Because we are connected to a CPSM WUI server, we benefit from the grouping of CICS regions into logical scopes, as provided by CPSM. Selecting a region, CICSplex or other logical scope in this view changes the scope of the other views, so that they show the information for that scope.

The top-right panel can show a variety of different views. Each view shows different aspects of the region or regions in the currently selected scope. There are a variety of views that are available that correspond to most of the resources that exist within a CICS region. There are several views that relate to Java in CICS.

The bottom-left panel also can show a variety of different views. The data displayed here often is dependent on what you select in the top-right panel. The properties tab, shown here, shows more information about the currently selected resource.

Many of the views have context menus (brought up by selecting the right-mouse button) that allow further information to be displayed about a given resource. If you have additional products (such as CICS Interdependency Analyzer or CICS Configuration Manager) installed then using these menu options, you can view additional information and apply changes to your CICS regions through those tools.

Do not overlook the status bar because it includes information, such as, the fact that we are connected to a server and provides name that we gave to the connection to the server.

You can customize the view by resizing, moving, or even removing panels. If you move these panels around, there is a menu option, **Window** → **Reset Perspective**, that restores the panels to their original positions and brings back any panels that you removed from the view.

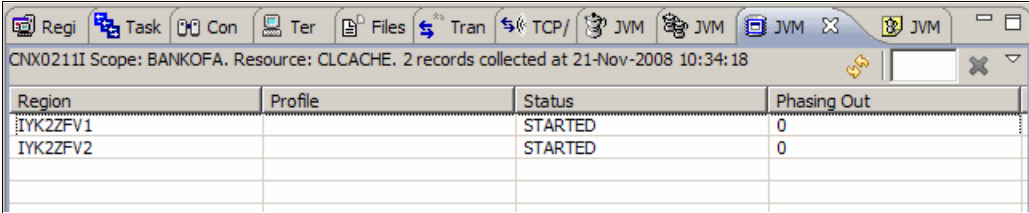
One of the more interesting menus is the ‘operations’ menu, which allows you to choose which of the CICS resources you want to see more information about. There is a selection of resource views that you can select, and we briefly review the Java related resources here. Under the **Operations** → **Java** menu, the following views are available:

- ▶ JVM Class Caches
- ▶ JVM Pools
- ▶ JVM Profiles
- ▶ JVM Status

We describe each of these views below.

JVM class caches

Class cache is used for JVM shared classes. This view, Figure 10-2, gives the status of the class cache and the number of JVMs being phased out.



Region	Profile	Status	Phasing Out
IYK22FV1		STARTED	0
IYK22FV2		STARTED	0

Figure 10-2 JVM Class Cache view

JVM pool

Each JVM runs on an MVS TCB, which is allocated from a pool of J8- and J9-mode open TCBs, which the CICS manages in the CICS address space.

This view, Figure 10-3 on page 237, gives you the status of the pool, the number of JVMs that are removed from the pool when they finished executing, and the number of pre-initialized JVMs.

If a JVM is not used by any application during the period of time that is specified in the IDLE_TIMEOUT option in its JVM profile, it becomes eligible for automatic termination.

Region	Status	Current	Phasing Out	Total
IYK2ZFBV1	✓ ENABLED	2	0	2
IYK2ZFBV2	✓ ENABLED	2	0	2

Figure 10-3 JVM Pool view

JVM profiles

Each JVM requires a profile that contains the properties that are needed for a particular JVM, for example, you can specify different JVM Profiles according to the type of program that is going to execute in the JVM. Figure 10-4 shows the JVM profile view.

Region	Name	Class Cache Status
IYK2ZFBV1	DFHJVMCD	NOCLASSCACHE
IYK2ZFBV1	DFHJMPC	CLASSCACHE
IYK2ZFBV1	DFHJMPC	NOCLASSCACHE
IYK2ZFBV2	DFHJVMCD	NOCLASSCACHE
IYK2ZFBV2	DFHJMPC	CLASSCACHE
IYK2ZFBV2	DFHJMPC	NOCLASSCACHE

Figure 10-4 JVM profile view

JVM status

This view, shown in Figure 10-5, displays the status of every JVM in the current scope.

Region	Profile	Name	Phasing Out Status
IYK2ZFBV1	DFHJMPC	17170743	NOPHASEOUT
IYK2ZFBV1	DFHJVMCD	17170997	NOPHASEOUT
IYK2ZFBV1	DFHJMPC	33948312	NOPHASEOUT
IYK2ZFBV1	DFHJMPC	50725517	NOPHASEOUT
IYK2ZFBV2	DFHJVMCD	393544	NOPHASEOUT
IYK2ZFBV2	DFHJMPC	33947828	NOPHASEOUT
IYK2ZFBV2	DFHJMPC	33948419	NOPHASEOUT
IYK2ZFBV2	DFHJMPC	50725469	NOPHASEOUT

Figure 10-5 JVM status view

10.2 CICS PA overview

CICS PA is a reporting tool that provides information about the performance of your CICS systems and applications to help you tune, manage, and plan your CICS systems effectively.

CICS PA is not an online monitoring tool. It produces reports and extracts using data that your system collects in MVS System Management Facility (SMF) data sets:

- ▶ CICS Monitoring Facility (CMF) performance class, exception class, and transaction resource class data in SMF 110 records
- ▶ CICS Transaction Server statistics data in SMF 110 records
- ▶ CICS Transaction Gateway statistics data in SMF 111 records
- ▶ System Logger data in SMF 88 records
- ▶ DB2 accounting data in SMF 101 records
- ▶ WebSphere MQ accounting data in SMF 116 records
- ▶ IBM Tivoli OMEGAMON XE for CICS on z/OS (OMEGAMON XE for CICS) data in SMF 112 records, containing transaction data for Adabas, CA-Datcom, CA-IDMS, and Supra database management systems

CICS PA can help:

- ▶ System Programmers to track overall CICS system performance and evaluate the results of their system tuning efforts
- ▶ Application Programmers to analyze the performance of their applications and the resources they use
- ▶ Database Administrators to analyze usage and performance of database systems, such as IMS and DB2
- ▶ WMQ Administrators to analyze usage and performance of their WebSphere MQ messaging systems
- ▶ Managers to ensure that transactions are meeting their required Service Levels and measure trends to help plan future requirements and strategies

CICS PA reports all aspects of CICS system activity and resource usage, including:

- ▶ Transaction response time
- ▶ CICS system resource usage
- ▶ Cross-system performance, including multi-region operation (MRO) and advanced program-to-program communication (APPC)
- ▶ CICS Business Transaction Services (BTS)
- ▶ CICS Web Support
- ▶ External subsystems, including DB2, IMS, and WebSphere MQ
- ▶ System Logger performance
- ▶ Exception events that cause performance degradation
- ▶ Transaction file and temporary storage usage

Rather than keeping large SMF data sets for reporting purposes, you can use CICS PA to load selected SMF records into a CICS PA historical database (HDB), optionally summarizing the records according to the time intervals that you require for reporting (such as hourly or daily). You can then use CICS PA to produce reports from the HDB instead of the SMF data sets. Loading selected and summarized SMF data into an HDB, you can accumulate the performance data that you want at the level of detail that you need for reporting over long periods, without requiring large amounts of storage or processing time.

In addition to producing formatted reports from SMF data sets or HDBs, CICS PA can extract data to DB2 tables or comma-separated value (CSV) text files. You can then develop your

own custom reports using DB2 SQL queries or download CSV files to your PC, where you can view and manipulate the data using PC-based spreadsheet applications, such as Microsoft Excel®.

CICS PA provides both an interactive ISPF dialog interface and a batch command interface. You can use either of these interfaces to request your reports and extracts. The ISPF dialog interface uses your interactive input to prepare JCL for the batch command interface. If you prefer to work directly with a command interface rather than an interactive interface, then you can use the ISPF dialog interface to prepare JCL that you can save and use as a starting point, and then edit the JCL later without using the ISPF dialog.

Figure 10-6 shows the SMF record types that CICS PA can read, and the output formats that it can produce.

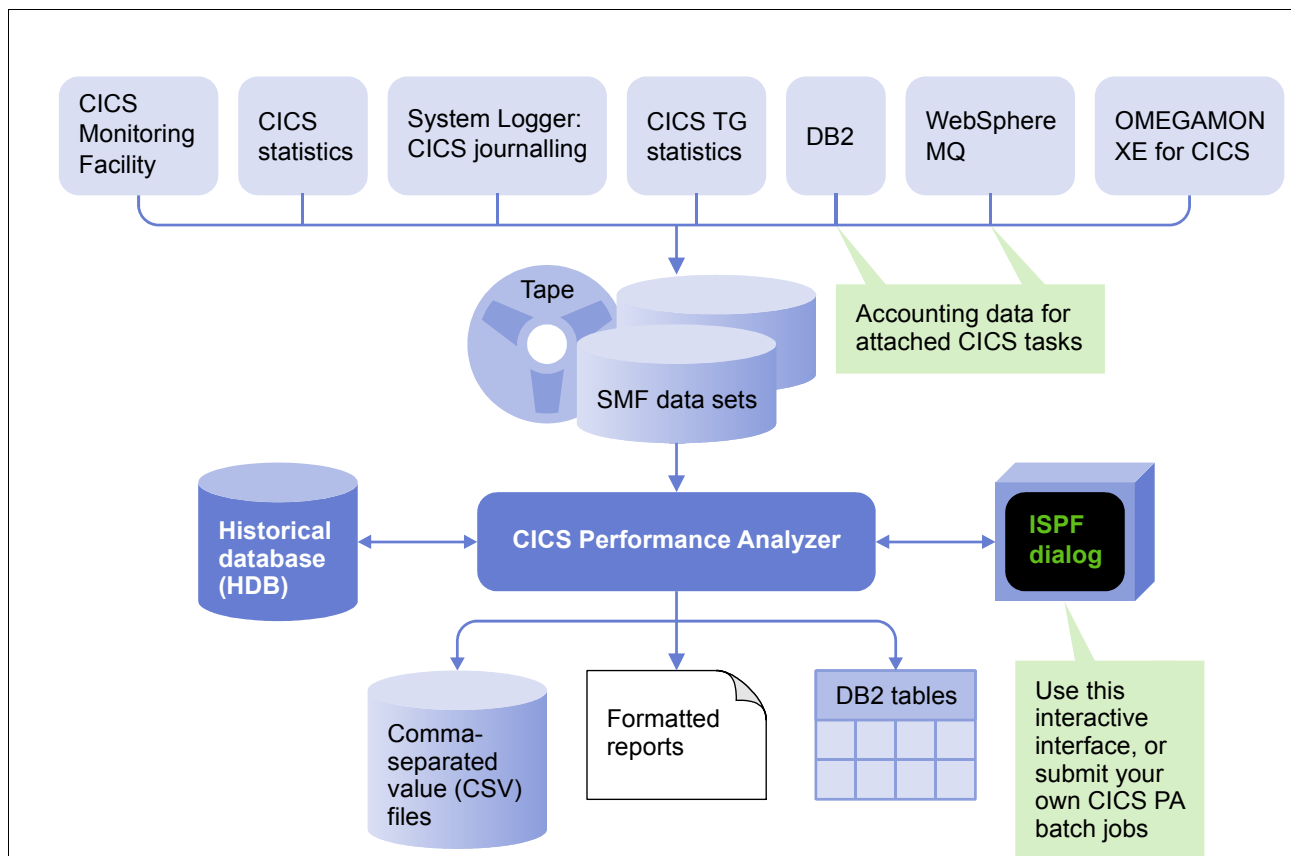


Figure 10-6 Overview of CICS PA inputs and outputs

Figure 10-7 on page 240 shows the primary option menu of the CICS PA ISPF dialog.

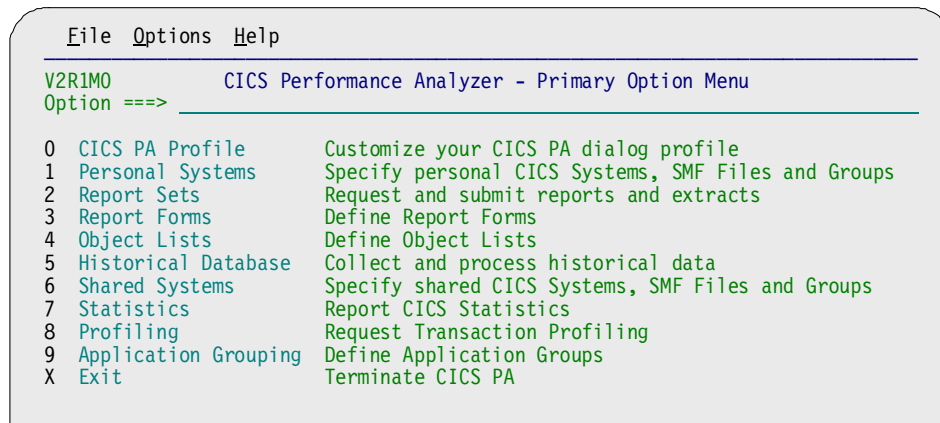


Figure 10-7 CICS PA primary option menu

Primary menu option 2, Report Sets, displays a list of the formatted reports that you can select, which are organized by category. Figure 10-8 on page 241 shows a sample report set. The Active column indicates the reports that this particular report set contains. You can generate and submit JCL to run reports individually in separate batch jobs, or you can submit an entire report set in a single batch job and perform a single pass over the input data.


```

V2R1M0      03:59:48 11/05/2008      CICS Performance Analyzer      Page 1
                                         Personal Systems Take-up from SMF

CPA2012I  Processing started for SMF file SMFIN001
CPA2017I  SMF records for System SC66 start at 11/05/2008 07:01:00.05
CPA2014I  CMF record for CICS system found, APPLID=A6POC3C1 Release=6.5.0
CPA2014I  CMF record for CICS system found, APPLID=A6POS3C2 Release=6.5.0
CPA2016I  MVS System Logger record found, System=SC66LOGR
CPA2013I  Processing ended for SMF file SMFIN001 - 3 system(s) found
CPA2000I  Personal Systems Take-up processing has completed, RC=0

```

Figure 10-9 Take-up output

Before running the first report, we chose a Report Form. We chose to use the CICS PA provided sample forms that are related to JVM. To bring these to our forms data set, on the CICS PA Primary Option Menu, we chose option 3 to go to the Report Forms panel. On this panel, we moved the cursor to the action bar under the Samples option and pressed Enter. We selected option 1 to populate the Report Forms data set with the sample forms, as shown in Figure 10-10.

```

Sample Report Forms   Row 77 to 89 of 156

Command ==>                               Scroll ==> PAGE

Select one or more Sample Report Forms then press EXIT.

Name      Type      Description
JVMLST    LIST      Java Virtual Machine Analysis
JVMSUM    SUMMARY   Java Virtual Machine Analysis
OMDLMLST  LIST      OMEGAMON Database Limit Warnings
OMOEMLST  LIST      OMEGAMON Third Party Support
OMOEMSUM  SUMMARY   OMEGAMON Third Party Support
OMRLMLST  LIST      OMEGAMON Resource Limit Warnings
PCLST     LIST      Program Request Activity
PCSUM     SUMMARY   Program Request Activity
PC3LST    LIST      Program Request Channel Activity
PC3SUM    SUMMARY   Program Request Channel Activity
PGAPLSUM  SUMMARY   Transactions by Application Prog
PGDPLSUM  SUMMARY   DPL Program Usage by Connection
PGUSESUM  SUMMARY   Transactions by Initial Program

```

Figure 10-10 Sample Report Forms pop-up panel

We scrolled down the list and selected the JVMLST and JVMSUM sample forms. We pressed F3 to return to our private Report Forms panel where we saw these forms included in the list.

We first produced the LIST report. We returned back to the Primary Option Menu, where we selected 2 to go the Report Set panel. On the Report Set panel, we entered NEW JVML to create a new Report Set. On the EDIT Report Set panel, we entered S next to the List option

in the category Performance Reports. On the Performance List Report panel, Figure 10-11, we entered the APPLID and image name as generated by the take-up, specified to use the JVMLST Report Form, and entered a report title.

File Systems Options Help	

JVML - Performance List Report	
Command ==>	
System Selection:	Report Output:
APPLID . . A6POC3C1 +	DDname LIST0001
Image . . SC66 +	Print Lines per Page . . (1-255)
Group . . +	
Report Format:	
Form . . . JVMLST +	
Title . . LIST report of JVM related fields	
Selection Criteria:	
Performance	

Figure 10-11 Performance List Report selections

We returned to the Edit Report Set panel where we ran the List report. We entered the time interval for our report, as shown in Figure 10-12 on page 244, and then submitted the report.

```

File  Systems  Options  Help
-----
                                Run Report Set JVML

Command ==>

Specify run options then press Enter to continue submit.

System Selection:

CICS APPLID . . A6POC3C1 + Image . .          + Group . .          +
DB2 SSID . . .      +      Image . .          + Group . .          +
MQ SSID . . . .      +      Image . .          + Group . .          +
Logger . . . .      +      Image . .          + Group . .          +

Override System Selections specified in Report Set

                                sssss Report Interval ssssss

Missing SMF Files Option:                                YYYY/MM/DD  HH:MM:SS.TH
1  1. Issue error message                                From 2008/11/05  01:00:00.00
    2. Leave DSN unresolved in JCL                        To   2008/11/05  03:00:00.00
    3. Disregard offending reports

Enter "/" to select option
/  Edit JCL before submit

```

Figure 10-12 Time interval selection for JVML report

Figure 10-13 shows the Performance List report.

CICS Performance Analyzer Performance List														
V2R1M0					CICS Performance Analyzer Performance List									
LIST0001 Printed at 22:09:39 11/10/2008					Data from 01:59:51 11/05/2008					APPLID A6POC3C1				
LIST report of JVM related fields										Page 1				
Tran	Userid	TaskNo	Stop	Response	Dispatch	User	CPU	J8 CPU	J9 CPU	JVM Elap	JVMITime	JVM Meth	JVMRTTime	JVM Susp
			Time	Time	Time		Time	Time	Time	Time	Time	Time	Time	Time
CWBA	CICSUSER	76078	02:33:49.958	.0015	.0014	.0013	.0000	.0011	.0011	.0011	.0000	.0011	.0000	.0001
CWBA	CICSUSER	76080	02:33:51.482	.0027	.0020	.0017	.0000	.0014	.0020	.0000	.0020	.0000	.0000	.0004
CWBA	CICSUSER	76082	02:33:52.136	.0016	.0014	.0014	.0000	.0011	.0011	.0000	.0011	.0000	.0000	.0001
CWBA	CICSUSER	76084	02:33:55.066	.0064	.0062	.0060	.0000	.0057	.0059	.0000	.0059	.0000	.0000	.0001
CWBA	CICSUSER	76086	02:33:55.819	.0017	.0015	.0015	.0000	.0011	.0012	.0000	.0012	.0000	.0000	.0001
CWBA	CICSUSER	76088	02:34:04.636	.0134	.0118	.0066	.0000	.0044	.0111	.0000	.0110	.0000	.0000	.0008
CWBA	CICSUSER	76091	02:34:05.245	.0017	.0015	.0014	.0000	.0011	.0012	.0000	.0012	.0000	.0000	.0001

Figure 10-13 The Performance List report

Conclusion

CICS Performance Analyzer for z/OS is a good reporting tool that helps you tune, manage, and plan your CICS systems in an efficient way. At the same time, CICS Performance Analyzer for z/OS supports JVM as well. Using CICS PA, you can analyze utilization and response time including JVM, KEY8 CPU, J8 CPU, and so on.

You can download CICS PA SupportPac CP12 from the Web at no charge:

<http://www.ibm.com/support/docview.wss?uid=swg24011321>

10.3 CICSplex System Management

CICSplex SM (commonly known as CPSM) is part of CICS Transaction Server and using it you can manage multiple CICS systems from a single control point. Enterprises who use CICSplex SM range from those who have 10 CICS systems to those that run hundreds of CICS systems.

It is not our intention in this section to discuss all of the features and advantages of using CICSplex SM. For complete information, access the CICS Infocenter that is applicable to the level of CICS Transaction Server that you installed and refer to the book entitled CICSplex SM Concepts and Planning.

Using CICSplex SM you can:

- Manage resources using Business Application Services (BAS)

BAS is an alternative to resource definition online (RDO) that enables you to manage CICS resources in terms of the business functions they belong to rather than their location in the CICSplex.

Using BAS has a number of advantages over RDO:

- The process is similar to RDO with a choice of interfaces
- Logical scoping allowing you to handle resources in terms of business application
- A common repository for all the resources in a CICSplex
- Reduced number of resource definitions leading to a consistency in definitions
- You can add resources to a region simply by adding it to the groups where the application executes
- You can direct CICSplex SM commands to a scope that matches the application instead of an arbitrary group of regions

- Manage Workloads

CICSplex SM workload management optimizes processor capacity in your enterprise by dynamically routing transactions and programs to which ever CICS region is the most appropriate at the time, taking into account any transaction affinities that might exist.

To achieve true workload balancing, design and write applications so that transaction affinities are either removed or kept to an absolute minimum.

There is a big difference between workload distribution and workload balancing. Workload distribution is distributing the work around the enterprise without necessarily taking into consideration how busy the target regions are and is often implemented using a “round robin” approach where target CICS regions are selected sequentially. Workload management can achieve workload balancing across the enterprise, provided the workload is actually high enough to be managed. This distinction is often a source of confusion and needs to be understood.

There are two algorithms that CICSplex SM uses to achieve workload balancing:

- The Queue algorithm

CICSplex SM sends work to the region that:

- Has the shortest queue of work waiting to be processed
- Is least likely to abend or meet conditions, such as SYSDUMP or TRANDUMP

- The Goal algorithm

CICSplex SM sends work to the region that:

- Meets the average response time goals set for it using the Workload Manager component of z/OS
- Is least likely to abend or meet conditions such, as SYSDUMP or TRANDUMP

When using the Goal algorithm you need to be aware that if one particular CICS region is meeting the goals set for the transaction the likelihood is that CICS Plex SM will continue sending work to this region until a condition occurs or the response time goal is not met.

- Exception Reporting using real-time analysis (RTA)

RTA provides automatic, external notification of conditions in which you might have expressed an interest, for example, if a FILE must always be ENABLED, the RTA component sends out warning messages whenever it finds that the file is not ENABLED.

The alerts can be sent either to the console or NetView or both:

- System Availability Monitoring (SAM)

This function monitors CICS systems during their planned hours of availability. If any of a set of predefined conditions occurs while the systems are being monitored, CICSplex SM sends out external notifications at the start of the condition and again when it is resolved.

- MAS resource monitoring (MRM)

You can use this to monitor the status of any specific or generic CICS resource and be informed when the state changes from a specified state, for example, when a critical FILE is CLOSED.

- Analysis point monitoring (APM)

When a resource is monitored using MRM in a number of regions there is a possibility that multiple notifications are sent when the state changes from the expected state. This function can be used to combine these messages into a single notification.

- Collecting statistics using CICSplex SM monitoring

CICSplex SM monitoring supports the collection of performance-related data at user-defined intervals within a set of CICS systems and can be used instead of the CICS Monitoring Facility (CMF)

The CICSplex SM Web User Interface

The CICSplex SM Web User Interface (WUI) is a customizable, platform-independent interface for your Web browser.

It is supplied with a set of linked menus and views to facilitate all of your system management tasks. You can customize the WUI to reflect your business procedures and the needs of individual users.

The CICSplex SM Web User Interface uses a frame-based interface.

- ▶ The Navigation Frame appears on the left of the display and contains items that you can use to display a menu or to view or perform an action.
- ▶ The Work Frame is the area where data is presented to you for interaction. Using this frame, you can set the Refresh rate and request a refresh by clicking the Refresh button.
- ▶ The Assistance Frame contains the product name, an icon linking to the Web User Interface Help, the IBM logo, and the Go back to start, Go back to last menu, and Go back icons.

For full details about the CICSplex SM Web User Interface, read the CICSplex SM Web User Interface Guide applicable to your installation.

The initial view

After you successfully logon to the Web User Interface, you are presented with this initial view of your CICSplex.

The Work Frame names the CMAS context, the Context, the Scope, and the views that are available to you and all of the links in the Navigation frame are closed.

Most views have the option to refresh the window by either setting a value for automatic refresh or a button for instant refresh.

Figure 10-14 on page 248 illustrates the Main menu.

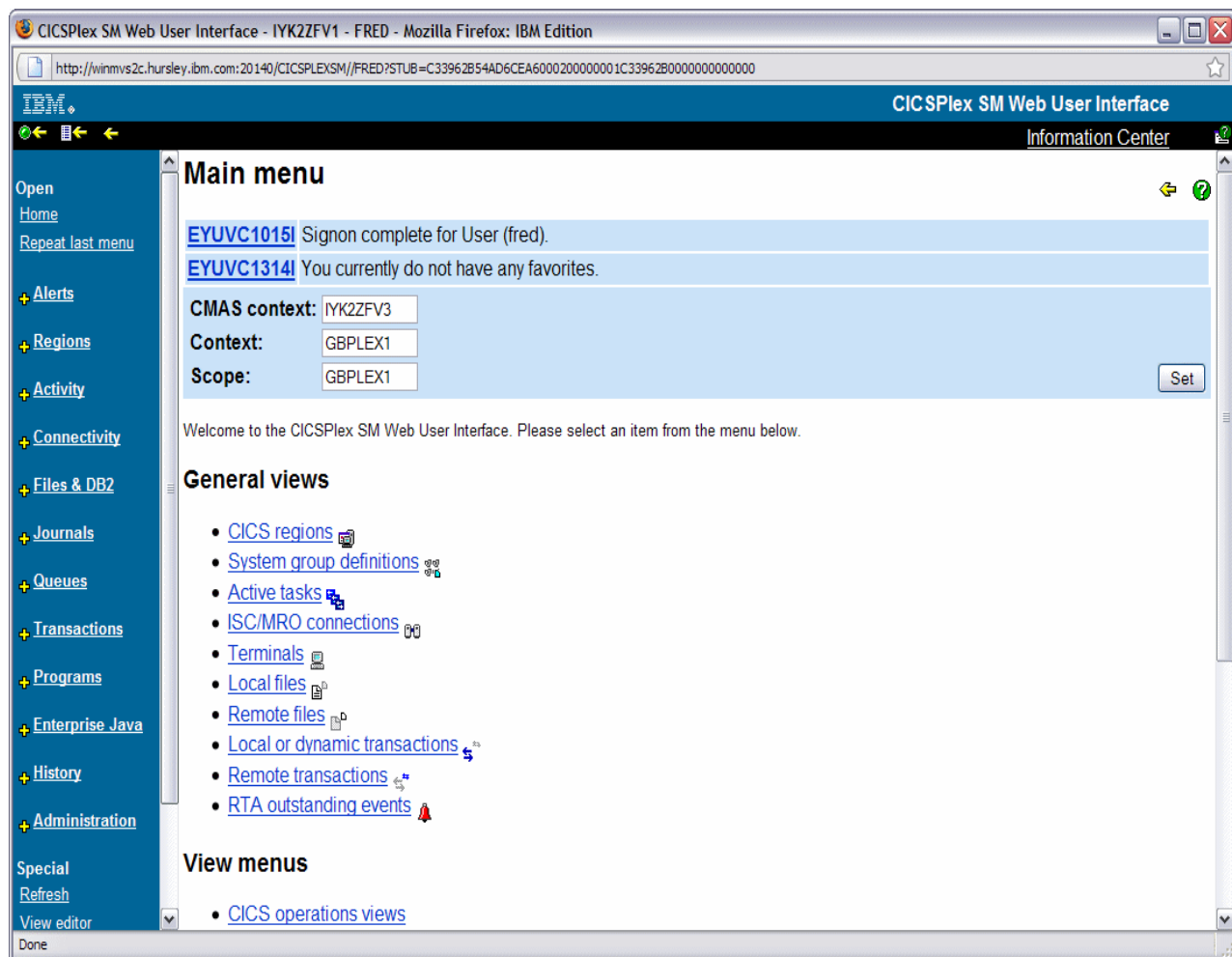


Figure 10-14 The CICSPlex SM supplied Main Menu

The Enterprise Java link

The Enterprise Java link is a clickable link in the navigation frame. Clicking it expands to give you links to the following views:

- ▶ Corbaservers
- ▶ CICS-deployed JAR files
- ▶ Enterprise Beans in CorbaServer
- ▶ Enterprise Beans in CICS-Deployed JAR files
- ▶ JVM Pool
- ▶ JVM Profile
- ▶ JVM Status
- ▶ JVM class cache

Figure 10-15 on page 249 shows the Enterprise Java link view.

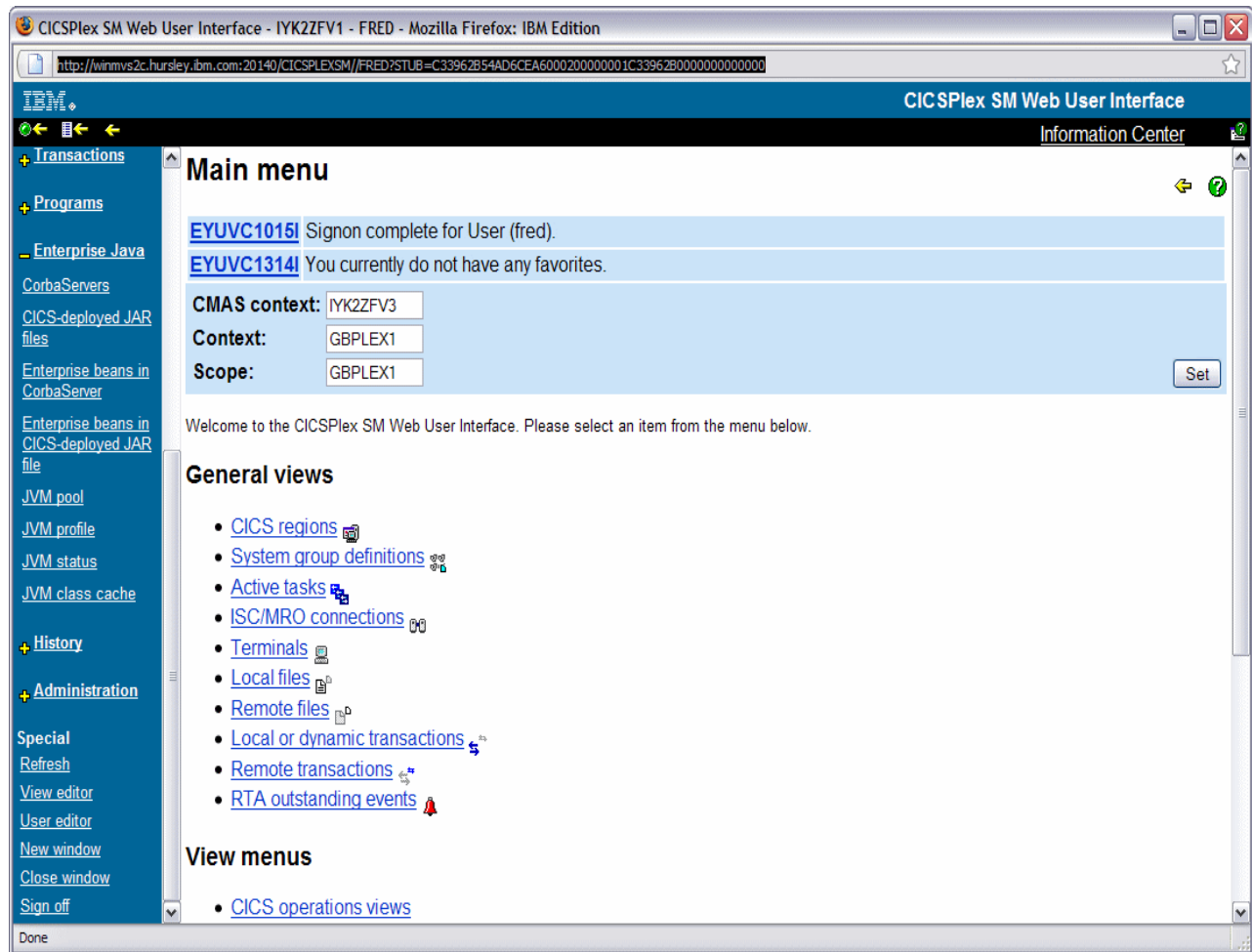


Figure 10-15 The Expanded Enterprise Java link

Corbaservers

The Corbaservers view, Figure 10-16 on page 250, shows all of the Corba servers that are defined in your CICSPlex.

The fields “CorbaServer name” and “TCP/IP host address” are links to views that give you detailed information about each resource.

There are four Action buttons available that you can use to manage the CorbaServer:

- ▶ Perform a CorbaServer scan to pick up any updated JARs in the zFS directory that is specified on the DJARDIR parameter of the CorbaServer definition
- ▶ Publish all Enterprise Java beans that are associated with the CorbaServer to the name server that is specified in the JVM properties file
- ▶ Retract all Enterprise Java beans that are associated with the CorbaServer from the name server that is specified in the JVM properties file
- ▶ Discard the CorbaServer resource

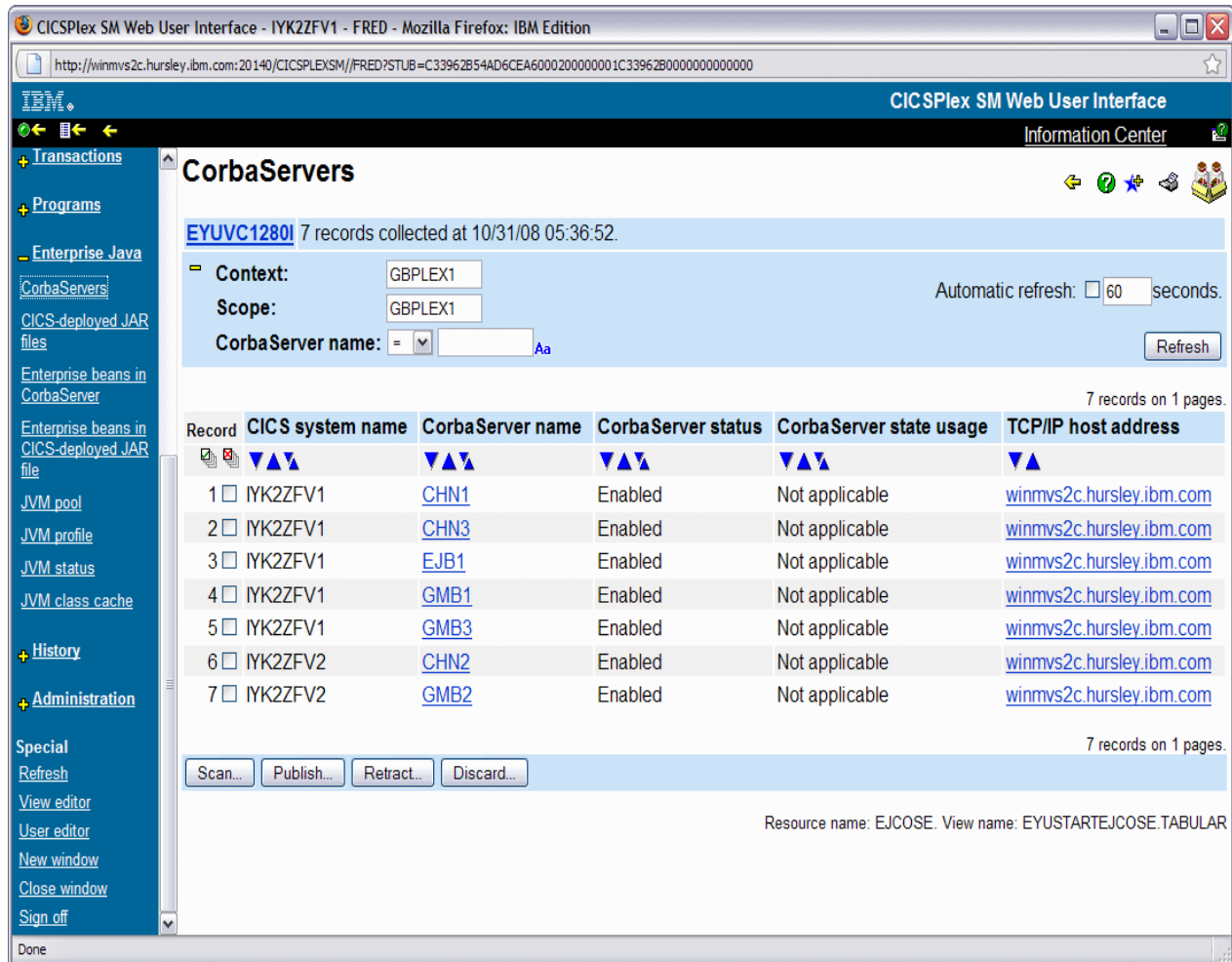


Figure 10-16 CorbaServers

CICS-deployed JAR files

These JAR files, shown in Figure 10-17 on page 251, are installed from the DJARDIR parameter that is specified on the CORBASERVER definition and are automatically installed by CICS after the CORBASERVER is installed.

The fields CICS-deployed Jar file, CorbaServer name, and Hierarchical file name (HFS) path are links to subsequent views that give detailed information about each resource.

There are three Action buttons available that allow you to manage the CICS-deployed JAR file:

- ▶ Publish all Enterprise Java beans that are associated with the CICS-deployed JAR file to the name server that is specified in the JVM properties file
- ▶ Retract all Enterprise Java beans that are associated with the CICS-deployed JAR file from the name server that is specified in the JVM properties file
- ▶ Discard the CICS-deployed JAR resource

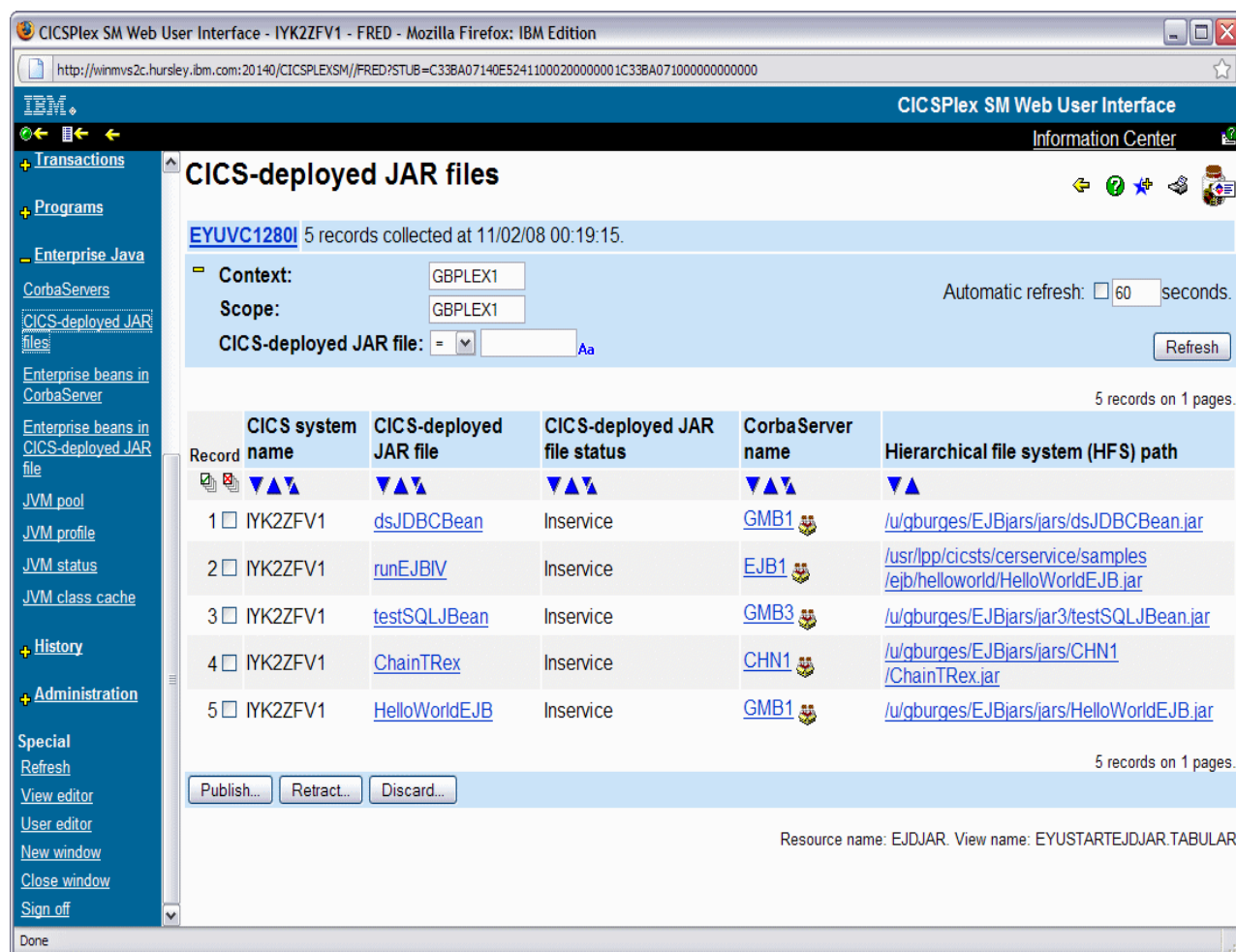


Figure 10-17 CICS-deployed JAR files

Enterprise beans in CorbaServer

These JAR files, Figure 10-18 on page 252, are installed from the DJARDIR parameter that is specified on the CORBASERVER definition and are automatically installed by CICS after the CORBASERVER is installed.

The fields CorbaServer name, Enterprise Bean name, and CICS-deployed JAR file are links to views that give detailed information about each resource.

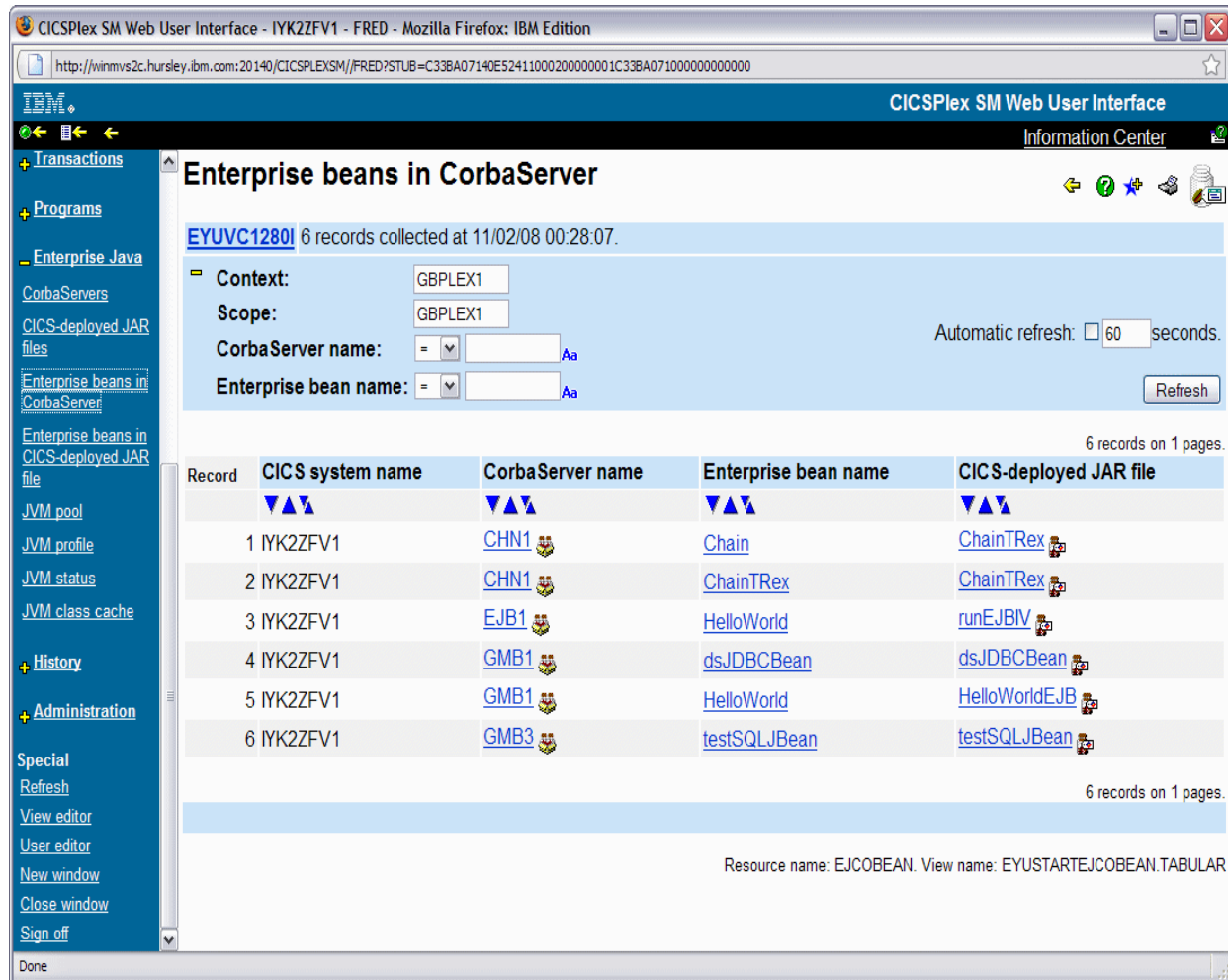


Figure 10-18 Enterprise beans in CorbaServer

Enterprise beans in CICS-deployed JAR file

This is a slightly different view to Enterprise Beans in CorbaServer. Here, the view lists each Enterprise bean in the CICS-deployed JAR file and the associated CorbaServer.

The fields CICS-deployed JAR file, Enterprise bean name, and CorbaServer name are links to views that give detailed information about each resource.

Figure 10-19 on page 253 shows the Enterprise beans in CICS-deployed JAR file window.

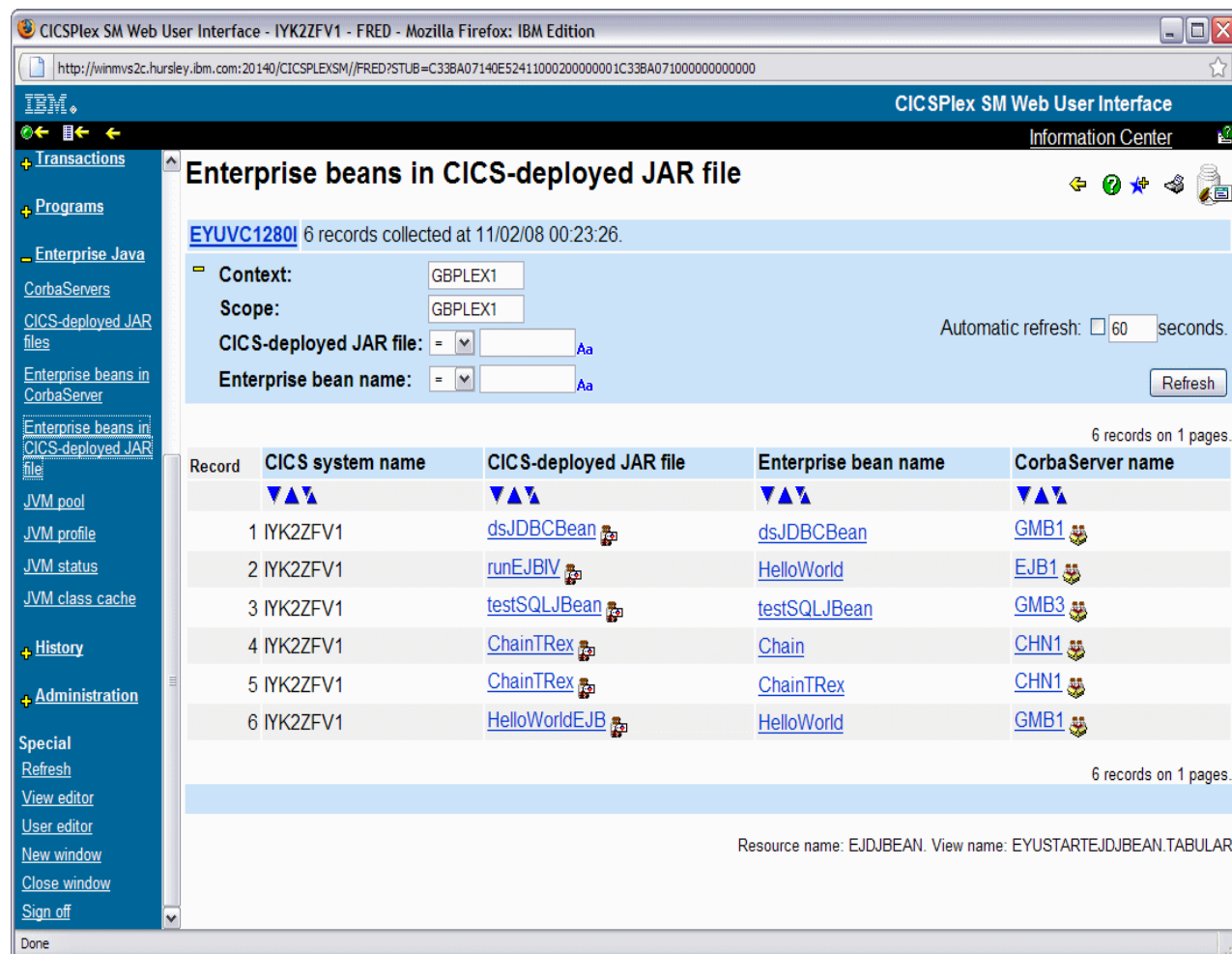


Figure 10-19 Enterprise Beans in CICS-deployed JAR file

JVM pool

Each JVM runs on an MVS TCB, which is allocated from a pool of J8- and J9-mode open TCBs, managed by CICS in the CICS address space.

The JVM pool view, Figure 10-20 on page 254, gives you the status of the pool, the number of JVMs to be removed from the pool when they finish executing, and the number of pre-initialized JVMs.

If an application does not use JVM during the period of time that is specified in the IDLE_TIMEOUT option in its JVM profile, it becomes eligible for automatic termination.

There are seven action buttons available that you can use to manage the JVM:

- ▶ Set Attributes to Enable or Disable the JVM Pool
- ▶ Enable: Enables the JVM Pool
- ▶ Disable: Disables the JVM Pool, which prevents new requests from being serviced from the pool. Currently, executing JVMs are allowed to terminate normally
- ▶ Phaseout: Marks all the JVMs for deletion and allows work that is currently running to complete

- Purge: Purges all tasks using the JVMs, and then terminates all JVMs using the pool
- Forcepurge: Force purges all tasks using the JVMs and then terminates all JVMs that are using the pool

The screenshot shows the 'JVM pool' configuration page in the CICSPlex SM Web User Interface. The page title is 'CICSPlex SM Web User Interface - IYK2ZFY1 - FRED - Mozilla Firefox: IBM Edition'. The URL in the address bar is 'http://winmvs2c.hursley.ibm.com:20140/CICSPLEXSM/FRED?STUB=C33962B54AD6CEA6000200000001C33962B000000000000000'. The page displays a table with 2 records for JVM pools, including columns for CICS system name, status, and number of virtual machines. The interface includes a left sidebar with navigation links and a top header with the IBM logo and page title.

Record	CICS system name	Status of Java virtual machine (JVM) pool	Number of Java virtual machines (JVM) for removal	Number of pre-initialized Java virtual machines
1	IYK2ZFY1	Enabled	0	3
2	IYK2ZFY2	Enabled	0	1

Buttons at the bottom: Set attributes..., Enable..., Disable..., Phase out..., Purge..., Force purge..., Start...
Resource name: JVMPOOL. View name: EYUSTARTJVMPOOL.TABULAR

Figure 10-20 JVM pool

JVM Profile

Each JVM requires a profile that contains the properties that are needed for a particular JVM, for example, you can specify different JVM profiles according to the type of program that is going to execute in the JVM.

The field “Name as used in a program definition” is a link to a view for detailed information.

Figure 10-21 on page 255 shows the JVM Profile view.

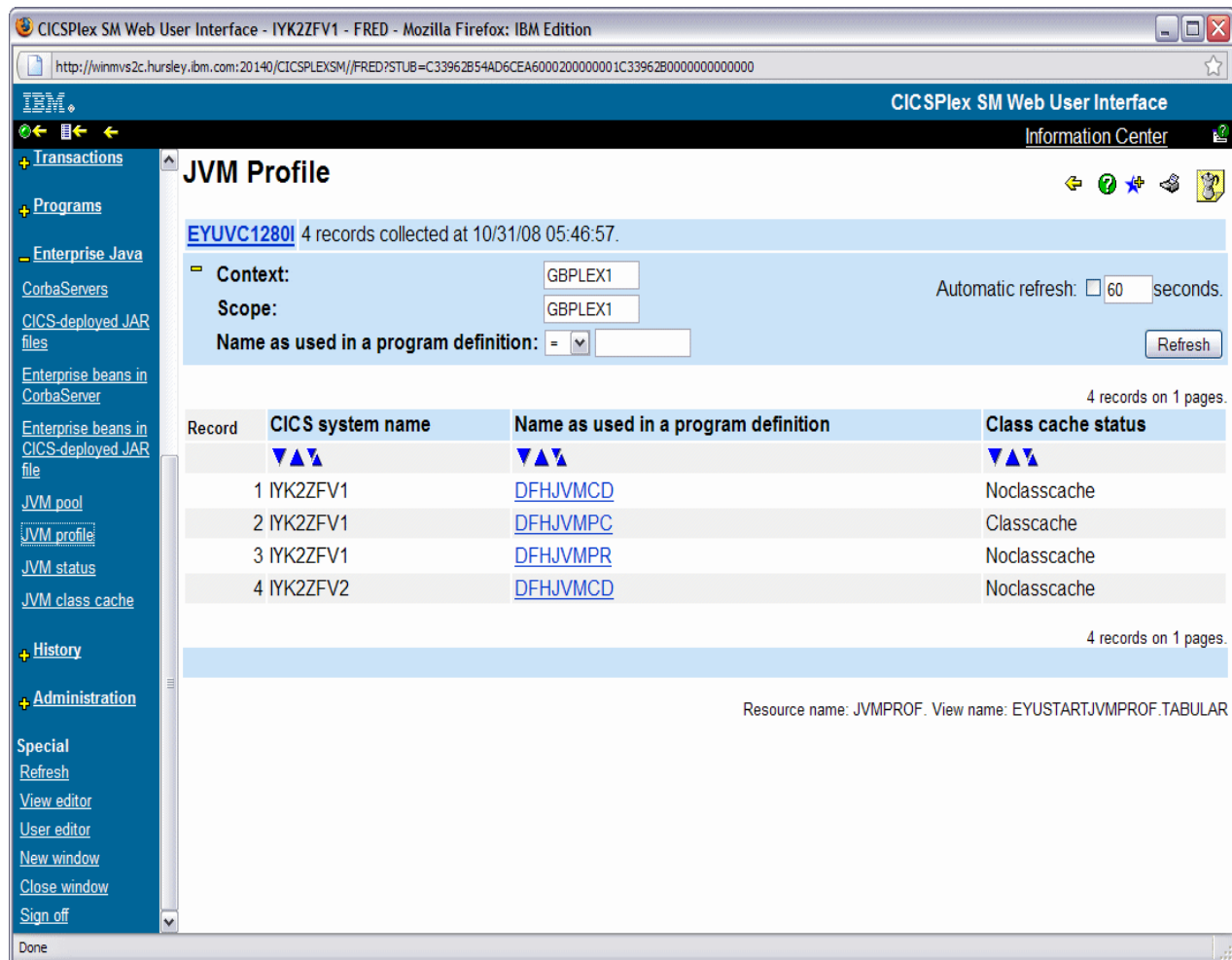


Figure 10-21 JVM Profile

JVM status

The JVM status view, Figure 10-22 on page 256, displays the status of every JVM in the CICSPlex. The Java Virtual Machine field is a link to a view that gives detailed information about the JVM, for example, the JVMProfile that is used to initialize the JVM.

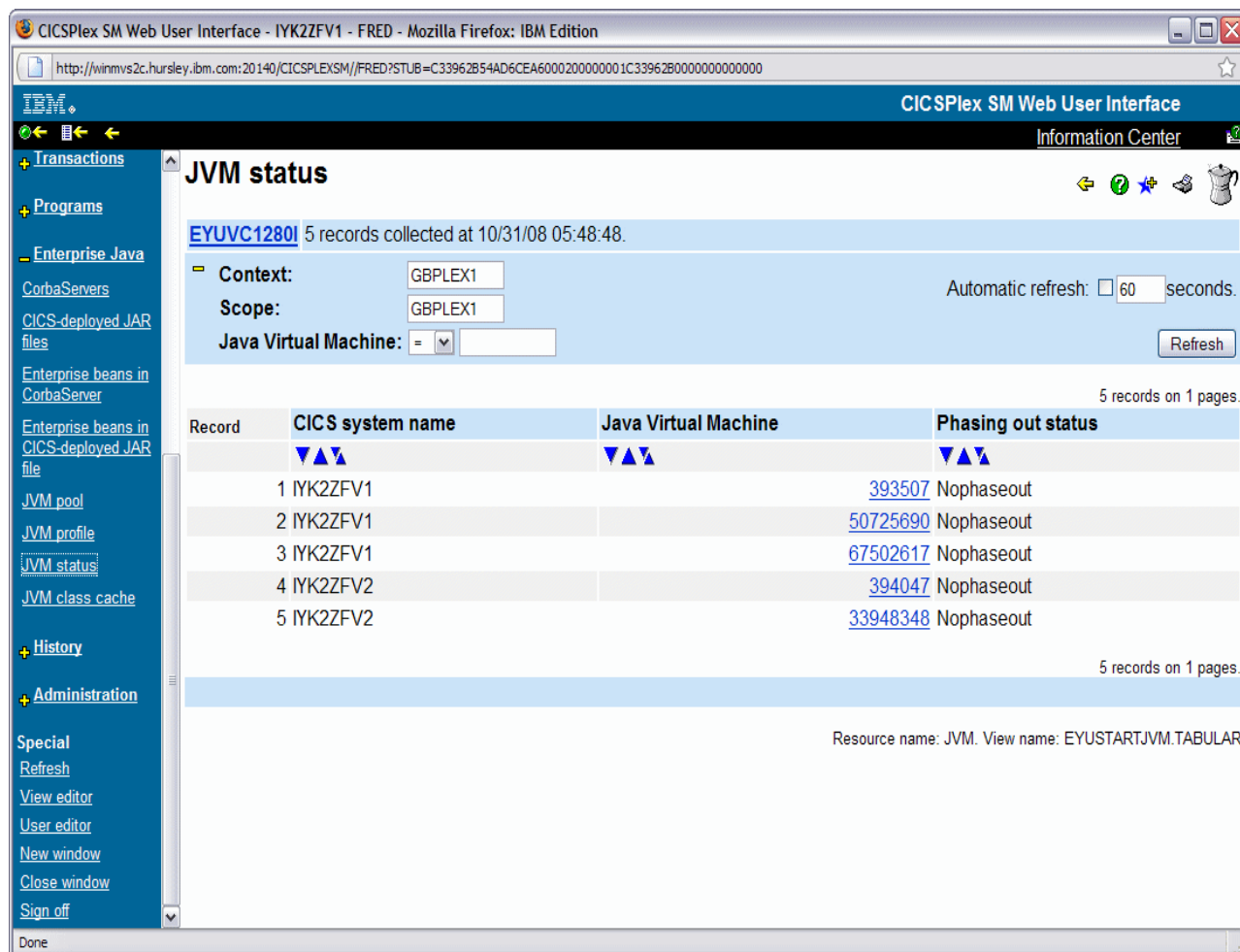


Figure 10-22 JVM status

JVM class cache

Class cache is used for JVM shared classes. The JVM class cache view gives the status of the class cache and the number of JVMs that are phased out.

The “CICS system name” is a link to a view that gives more information about the class cache, such as time started, size of the class cache, and amount of free space in the class cache.

There are five action buttons available that you can use to manage the JVM class cache:

- ▶ **Phaseout:** Marks all of the JVMs that are using the shared class cache for deletion, which allows work that is currently running to complete. The shared class cache is deleted when all JVMs that are using it are terminated
- ▶ **Purge:** Purges all JVMs using the shared class cache. The shared class cache is deleted when all JVMs using it are terminated
- ▶ **Forcepurge:** Force purges all JVMs using the shared class cache. The shared class cache is deleted when all JVMs using it are terminated
- ▶ **Start:** Creates a new class cache when the status is STOPPED
- ▶ **Reload:** Creates a new class cache when the status is STARTED

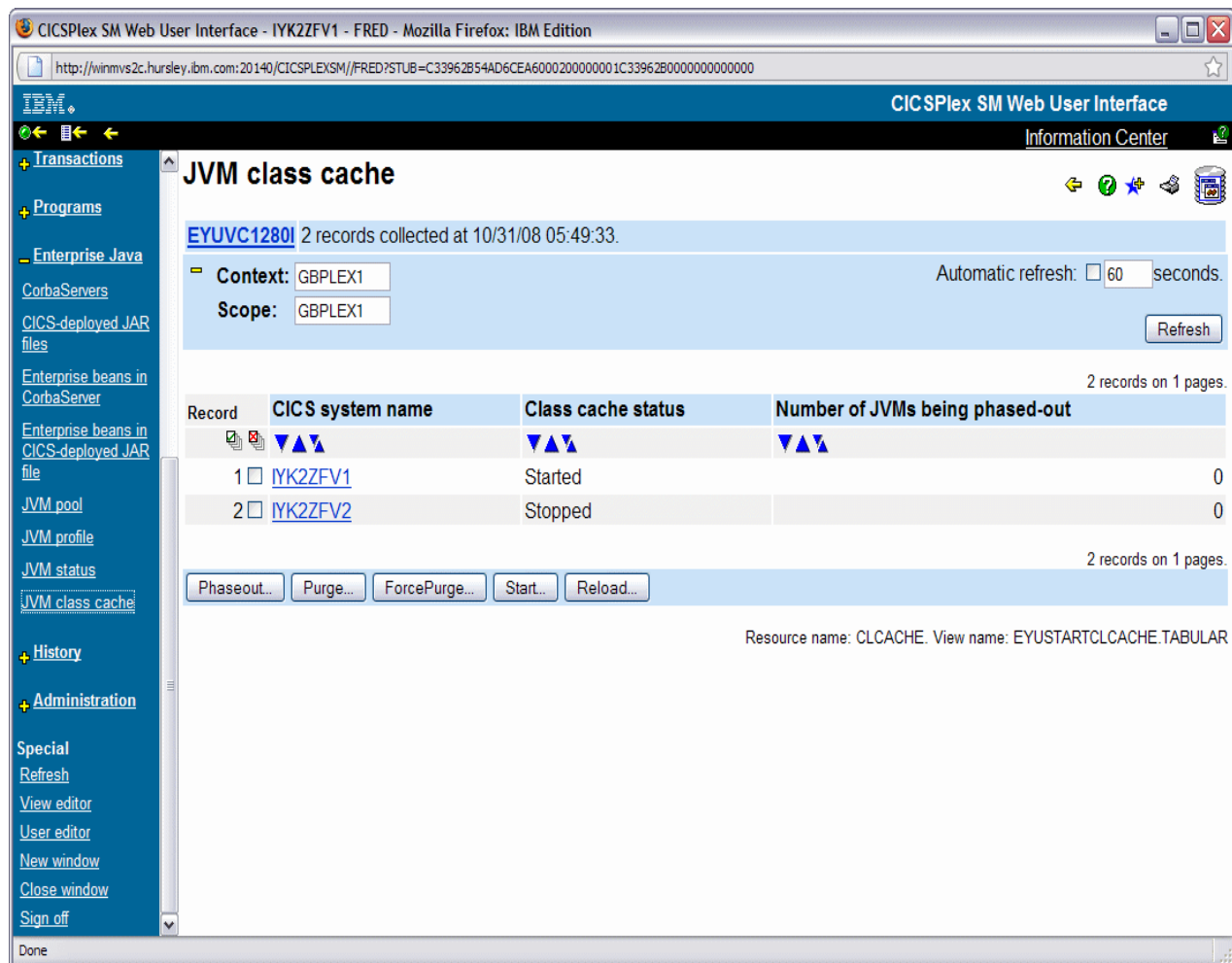


Figure 10-23 JVM class cache

10.4 OMEGAMON XE for CICS on z/OS

OMEGAMON XE for CICS on z/OS is the component of IBM Tivoli Monitoring that provides the capability to monitor CICS Transaction Server (TS) environments. It is installed as a monitoring agent in the IBM Tivoli Monitoring (ITM) framework. It provides workspaces and situations that provide alerts when components of the CICS TS environment are not meeting the expected performance parameters.

As a component of ITM, OMEGAMON XE for CICS also provides the capability to combine CICS monitoring with that of other monitored components of the enterprise. It uses Dynamic Workspace Linking (DWL) to allow a user to switch between OMEGAMON XE for CICS and other OMEGAMONs, such as OMEGAMON XE for DB2 on z/OS, which gives users the capability to follow transactions to identify the route of a problem.

OMEGAMON XE for CICS on z/OS also provides a 3270 interface that provides the capability to perform in depth diagnosis of a CICS TS system.

In this section, we explain the major components of OMEGAMON XE for CICS and describe some of the major features. This is not a comprehensive description, but you can use it to gain an understanding of the product.

OMEGAMON XE for CICS components

For OMEGAMON XE for CICS to provide information relating to CICS Transaction Server, the following components must be configured on the z/OS image where the monitored CICS regions reside:

- ▶ An OMEGAMON XE for CICS Monitoring Agent
- ▶ An OMEGAMON II for CICS Menu System
- ▶ An OMEGAMON II for CICS CUA interface (optional)
- ▶ OMEGAMON for CICS component in monitored CICS regions
- ▶ Configuration of OMEGAMON II for CICS

In the next sections, we discuss the function and specific requirements of each of these components in more detail. For the purposes of this chapter, we assume that the ITM framework is installed and configured.

OMEGAMON XE for CICS Monitoring Agent

The OMEGAMON XE for CICS monitoring agent is the component that is responsible for responding to queries from the ITM framework for data relating to the CICS TS systems that are currently monitored. The agent obtains information by interfacing with OMEGAMON code that runs in the CICS address space. It also communicates directly with the Menu System started task (KOCCI) to query data that it maintains. The agent also has a Workload Manager (WLM) component that generates summaries of transaction performance against specified goals.

The monitoring agent provides the data that is required to populate the CICS workspaces on the TEP. Figure 10-24 shows an example of a workspace that is provided with the OMEGAMON XE for CICS on z/OS product.

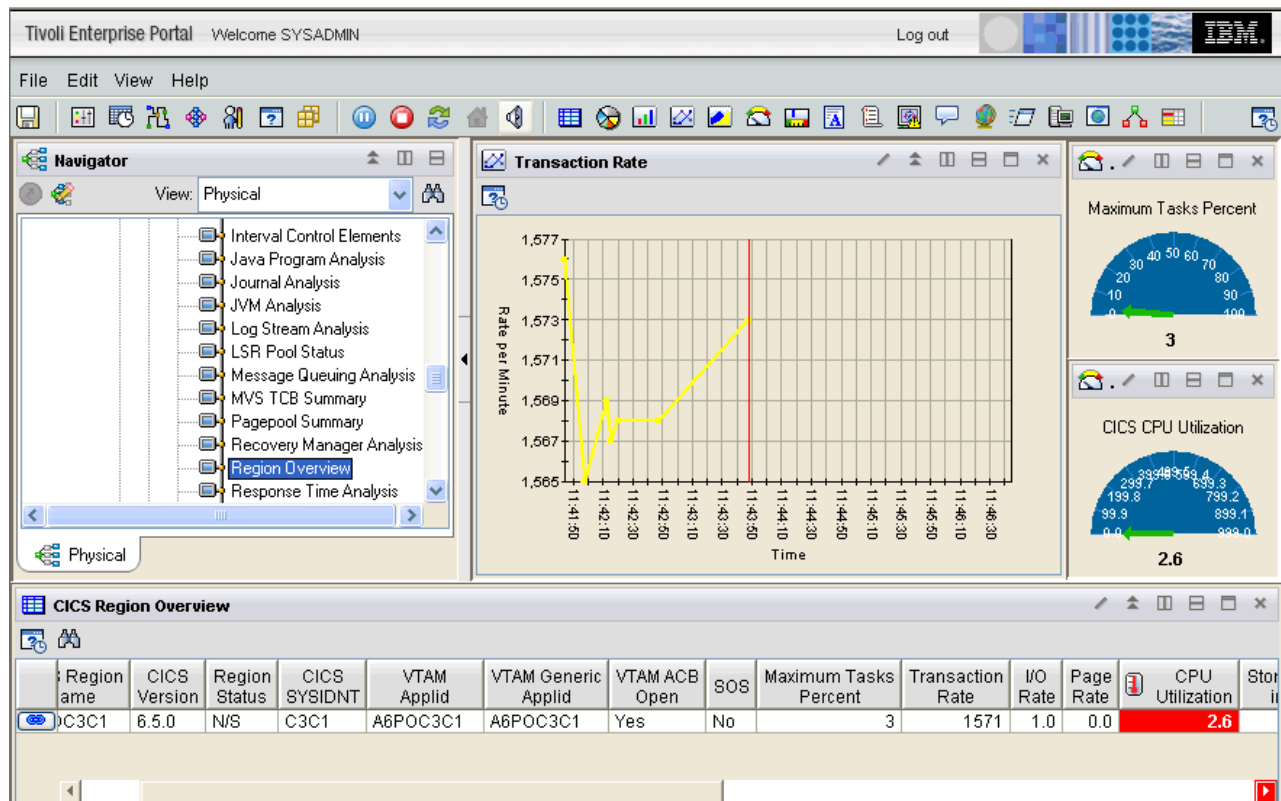


Figure 10-24 OMEGAMON XE for CICS region overview

An agent must be deployed on each LPAR where CICS TS regions run. It is recommended that the OMEGAMON CICS agent run its own address space.

The OMEGAMON XE for CICS agent is configured using the Installation and Configuration Assistance Tool that is provided with ITM.

Figure 10-25 shows the configuration page for the OMEGAMON XE for CICS agent.

```

----- CONFIGURE IBM TIVOLI OMEGAMON XE FOR CICS ON Z/OS / RTE: SC66 -----
OPTION ==>

Perform the appropriate configuration steps in order:

  I Configuration information (What's New)

  1 Register with local TEMS (required if the Agent
    will connect to the TEMS in this RTE.)

  2 Specify configuration parameters

Agent address space configuration:
  3 Specify Agent address space parameters
  4 Create runtime members
  5 Configure persistent datastore (in Agent)

  6 Complete the configuration

Note: This Agent is running in its own Agent address space.

F1=Help F3=Back F5=Advanced

```

Figure 10-25 Configuration page for OMEGAMON XE for CICS on z/OS

Notes about the OMEGAMON configuration page options:

- Option 1, Register with local TEMS: Adds application support files to the TEMS. Here is a summary of when you are required to select this option:
 - If the agent you are configuring reports to a TEMS that is configured in the same (that is, local) RTE.
 - If your hub TEMS is running on z/OS but the agent reports to a remote TEMS, register the monitoring agent with both the remote TEMS it directly reports to and the hub TEMS.
 - If the agent reports to the hub TEMS directly, you only need to register it one time. If the hub TEMS is configured in a separate RTE from the agent, select that RTE for configuration of the OMEGAMON XE agent, but only select option 1 (Register with local TEMS). None of the other configuration items need to be completed unless you are also going to define an OMEGAMON XE agent in the same RTE as the Hub TEMS.
 - If the agent reports directly to the Hub TEMS, but this TEMS is configured in a different RTE from the agent, you must select that RTE for configuration and then only select option 1 (Register with local TEMS) for this configuration. None of the other selected agent configuration items that follow should be performed.

In our configuration, where we want to report directly to a remote TEMS in the same RTE as the monitoring agent and our hub TEMS is installed on another operating system, we must register our monitoring agent just one time with the remote TEMS.

- In option 2, Specify configuration parameters: Shown in Figure 10-26 on page 260, we specify the configuration options that are unique to OMEGAMON XE for CICS on z/OS.

```

----- SPECIFY CONFIGURATION PARAMETERS -----
OPTION ==>

Complete the items on this panel.

WLM block allocation      ==> 236              (10-524287 blocks)
WLM collection interval  ==> TEP              (TEP or 1 minute)

Enter=Next  F1=Help  F3=Back  F5=Advanced

```

Figure 10-26 OMEGAMON XE for CICS on z/OS product specific parameters

The *WLM block allocation* specifies the number of 4096 byte blocks that are allocated to the WLM data space. This data space stores information while summaries are created. Although the usage of this storage is stable for a given workload, it is not easy to predict. It is recommended that the number start at 236. If the message KCP0244 is produced by the agent, this indicates that the value must be increased.

The WLM collection interval controls the time span of Service Level Analysis displays at the TEP. OMEGAMON WLM accumulates data after a minute in two sets of records, the "5 minute records" and the "interval records". The "5 minute records" are displayed at the TEP. The "interval records" are used for history (Persistent Data Store).

Calculations for values, such as average response time, are performed every five minutes and at the end of the history interval.

Specifying a value of 1 transforms the "5minute records" to "1minute records". Starting WLM with a frequency interval value of 1 results in the TEP displaying Service Level data collected at the last "1minute" interval.

Specifying a value of TEP results in the TEP display of the "5minute records" using the values specified via TEP.

The acceptable values for the INTERVAL keyword are 1 or TEP. You can specify values other than 1 using the SLA CICS view of the TEP.

Options 3, 4, and 5 are common to all of the OMEGAMON XE monitoring products. These steps are required to be completed if you are configuring the agent to run in its own address space.

- ▶ Option 3, Specify Agent address space parameters: Where you specify the communication values for reporting to the hub and the "self-describing" values for configuring up your agent address space, such as security, locale, and so on.
- ▶ In option 4, Create runtime members: A batch job is generated that updates the user parameter libraries with the values you specified as part of configuration options 2 and 3. After submitting this job check, the return code produced no errors.
- ▶ Option 5, Configure persistent data store (in Agent): Allocates the persistent data store files and starts up parameters as part of the agent address space. During the persistent data store configuration, you are asked to enter JCL jobcard information. Note that this jobcard information is shared by all persistent data store runtime JCL generated for this RTE.

If you have a TEMS defined in this RTE and want to run the agent in the local TEMS address space, you must select F5, and follow the advanced configuration options. This is not the recommended agent configuration, so we do not discuss it in this book.

- ▶ Option 6, Complete the configuration: Guides you through some manual configuration steps that must be performed outside of the Configuration Tool, such as copying the agent

started task to your PROCLIB and ensuring libraries are APF authorized. These steps are required to be completed prior to testing your configuration.

OMEGAMON II for CICS Menu System

The OMEGAMON II for CICS menu system consists of a started task, known as the common interface, and also referred to as the KOCCI. It provides a 3270 interface to allow monitoring of the CICS regions on the same LPAR. It provides an anchor to the OMEGAMON for CICS components that run in a CICS region. It is responsible for loading of the Global Data areas that control which features of OMEGAMON II are to be active. It also provides an environment for the subtasks that are required to provide Bottleneck Analysis, Response Time Analysis, and Transaction History collection.

The 3270 interface that the KOCCI provides gives the user a highly efficient mechanism to view specific information relating to a CICS region. The menu system allows you to see resource detail information and detailed information regarding the memory usage and DASD performance of the devices that the CICS region currently uses. The Menu system also allows authorized users to view and modify the storage in the CICS region.

Figure 10-27 shows the primary panel that the menu system provides.

```

      ZMENU      VTM      A6POC3C1 V560./C SC66 09/21/08 12:08:24
> PF1 Help/News/Index PF3 Exit PF18 Color PA2 REGION STATUS
=====
>
      OMEGAMON II FOR CICS PERFORMANCE MONITOR SYSTEM
>
      Enter a selection letter on the top line.
> W REGIONS ..... List CICS regions, switch monitoring
> V OVERVIEW ..... Performance overview
> R RESPONSE TIME ..... CICS and end-to-end response time monitoring
> E EXCEPTIONS ..... Current system problems and problems this session
> T TASKS ..... Task analysis
> H HISTORY ..... Historical, traced transaction viewing and selection
> B BOTTLENECKS ..... Resource contention (bottlenecks, impacts, enqueues)
> S STORAGE ..... Storage summary, violations, DSA, EDSA, PAM, subpools
> F FILES ..... CICS datasets, VSAM, LSR, string and buffer waits
> D DATABASES ..... DB2, DLI and MQ
> C CICS ..... CICS tables and control blocks
> M MVS ..... Operating system control blocks for CICS
> I I/O ..... DASD performance
> U UTILITIES ..... Utility functions and displays
> O CONTROL ..... Control options (response time, history, contention,
>                   database collectors, shutdown, SMF, trace)
> G GROUPS ..... Group definitions
> P PROFILE ..... Profile options and maintenance
=====
```

Figure 10-27 OMEGAMON II menu system interface

The menu system can only monitor CICS regions that are running on the same LPAR. For that reason, a KOCCI must be configured on each LPAR where CICS regions that are to be monitored can run.

OMEGAMON II for CICS CUA interface (optional)

The OMEGAMON II for CICS CUA interface provides a means of viewing the data provided by the Menu system that conforms to the Common User Access (CUA) standard.

The interface provides an easy point and shoot type access to the monitoring data. An example of the CUA Region Overview panel is provided in Figure 10-28 on page 262.

Actions GoTo Index Options Help KC2B01D6 Region Status Select one component with a / or an action code. S=Show details A=Analyze problems L=Control			09/21/08 12:57:27 PM Region: A6P0C3C1 + Auto(Off)		
Workloads		Resources		Alerts	
- AIDs	OK	- CPU	OK	- MRO/ISC	Idle
- ICEs	OK	- DASD	Idle	- Paging	OK
- Response	OK	- DB2	Idle	- Storage	OK
- Tasks	Crit	- DBCTL	Idle	- Tapes	Idle
- TranRate	Idle	- Files	OK	- TCP/IP	Idle
- UOWs	OK	- Journals	OK	- TempStor	OK
		- LSR	OK	- TranData	Warn
		- MQ	Idle	- Web	OK
				- Bottlnck	Idle
				- CICSloop	OK
				- Dumps	OK
				- Enqueues	OK
				- I/O Rate	Idle
				- VTAM ACB	OK
				- ZRF	Idle

F1=Help F3=Exit F4=Prompt F5=Refresh F6=Console F10=Action Bar F11=Print
PA1=Switch

Figure 10-28 CUA region overview

The CUA interface is not a required part of the OMEGAMON CICS configuration. The menu system and OMEGAMON XE for CICS provide full functionality, regardless of the status of the CUA interface.

OMEGAMON for CICS component in monitored CICS regions

For OMEGAMON CICS to provide full functionality, you must make certain changes to the CICS regions that are to be monitored. OMEGAMON code needs to be installed and run in each CICS region to allow OMEGAMON to enable certain CICS Global User Exits (GLUEs) to allow the collection of data that is required for features, such as Transaction History, Service Level Analysis, Response Time Analysis, and some resource monitoring.

OMEGAMON II for CICS requires that a Program and a Transaction be defined, as shown in Figure 10-29.

```

DEFINE PROGRAM(KOCOME00) GROUP(OMEGAMON) CONCURRENCY(THREADSAFE)
EXECCKEY(CICS)

DEFINE TRANSACTION(OMEG) PROGRAM(KOCOME00) GROUP(OMEGAMON)
TASKDATAKEY(CICS)

```

Figure 10-29 RDO definitions for OMEGAMON CICS

CICS PLT changes

For OMEGAMON CICS to be configured to start and stop automatically, the CICS PLTs must be updated. Here are the PLT additions for installation:

DFHPLT TYPE=ENTRY,PROGRAM=KOCOME00

Specify the PLT additions, as shown, in the following tables:

- ▶ PLTPI (initialization) *immediately after* the DFHDELIM entry for CICS Transaction Server systems.
- ▶ PLTSD (shutdown) *before* the DFHDELIM entry.

CICS JCL changes

In this section, we provide the JCL changes that you can make.

Add a DD statement for RKANMOD and concatenate the Tivoli OMEGAMON II load library, RKANMOD, to DFHRPL:

```
//RKANMOD DD DISP=SHR,DSN=rhilev.RKANMOD
//DFHRPL DD DISP=SHR,DSN=.....
// DD DISP=SHR,DSN=rhilev.RKANMOD
```

In addition to the changes above there are some optional JCL changes that you can make:

- ▶ `//KOCGLBnn DD DUMMY`

This instructs OMEGAMON CICS that the Global Data area with the suffix *nn* is to be used for this CICS region.

- ▶ `//OCCIREQ DD DUMMY`

This instructs OMEGAMON to check for the presence of the KOCCI address space. If it is not found, a message OC0806 is issued asking the operator if this CICS region is to continue in this case.

It is possible to run more than one copy of a KOCCI or OMEGAMON CICS agents on an LPAR. If this is the case, you must indicate which address space is to monitor this CICS region. This is achieved by placing corresponding DD cards in the CICS region JCL and the KOCCI address space JCL or agent JCL.

If more than one KOCCI is required, place the following card in both the CICS region and the KOCCI JCL:

```
//RKC2XMnn DD DUMMY
```

Where *nn* is a number between 00 and 15. If no card is specified, it is the same as using the number 00. Only one KOCCI can be active on an LPAR for a given release with any one number.

If more than one agent is required, put the following card in both the CICS region and the AGENT JCL:

```
//RKCPXMnn DD DUMMY
```

Where *nn* is a number between 00 and 15. If no card is specified it is the same as using the number 00. Only one agent can be active on an LPAR with any one number

Configuring OMEGAMON II for CICS

You configure OMEGAMON II for CICS using the Installation and Configuration Assistance Tool that is provided with ITM. Figure 10-30 on page 264 shows the configuration page for OMEGAMON II for CICS.

```

----- CONFIGURE OMEGAMON II FOR CICS / RTE: SC66 -----
OPTION ==>

Perform these configuration steps in order:                    Last selected
                                                             Date      Time

  I Configuration information (What's New)

    1 Specify configuration values                            08/09/18  13:35
    2 Allocate additional runtime datasets                    07/08/22  15:19
    3 Create runtime members                                 08/08/28  16:55
    4 Complete the configuration                             08/09/11  22:08

Optional:

    5 Allocate/initialize task history datasets              08/09/11  21:35
    6 Manage CICS global data area(s)                       08/09/16  15:08
    7 Modify menu system command security                   08/08/28  16:56
    8 Install OMEGAMON Subsystem                            07/08/22  15:20

F1=Help  F3=Back

```

Figure 10-30 OMEGAMON II for CICS configuration panel

To configure OMEGAMON II for CICS:

1. Option 1, Specify configuration values: Walks you through the product-specific values for OMEGAMON II for CICS.

Figure 10-30 shows the first panel that is presented for option 1.

```

----- OMEGAMON II FOR CICS CONFIGURATION VALUES / RTE: SC66 -----
OPTION ==>

VTAM information:
  Maximum number of CUA users                ==> 99      (10-256)
  Enable ACF/VTAM authorized path            ==> N        (Y, N)

CUA security options:
  Specify security                           ==> RACF      (RACF, ACF2, TSS,
                                                             NAM, None)
  Function level security resource class      ==> $OMEG     (ACF2 max is 3 char)

Started task:
  End-to-End (ETE)                           ==> started_task_name

Advanced options:
  Maximum number of KOCCI users (UMAX)       ==> 99      (1-99)
  Copies of OMEGAMON II address spaces       ==> 1        (1-16)
  Fold CUA output to upper case              ==> N        (Y, N)
  Enable CUA simplified signon               ==> Y        (Y, N)
  Enable CUA WTO messages                   ==> N        (Y, N)

Enter=Next  F1=Help  F3=Back

```

Figure 10-31 OMEGAMON II configuration values

- a. Enter the name of the started task for the End-to-End component. Make sure that the name of the started task for End-to-End (ETE) is correct and is unique for this RTE. You can accept the defaults for the remainder of the values.
- b. When the values are complete, press **Enter** to reveal the panel in Figure 10-31.

```

----- OMEGAMON II FOR CICS CONFIGURATION VALUES / RTE: SC66 Row 1 from 48
COMMAND ==>

Modify the parameters below to suit your site's requirements.

ID: 00  Menu STC:  CANSOCO_  Menu applid: &SYSNAME.OCO_____
      CUA STC:   CANSO20_  CUA applid:  &SYSNAME.C20_____
                                CUA operator: &SYSNAME.C200_____
                                VTERM prefix: &SYSNAME.CO_____
                                Default CICS rgn: *_____
      Major node: &SYSNAME.C20N_____
Enter=Next F1=Help F3=Back F7=Up F8=Down

```

Figure 10-32 OMEGAMON II configuration values continued

In Figure 10-32, the started task names and applids are entered. In this case, we are using the z/OS system variables to prefix the applids, which makes it easier to configure environments for multiple LPARs.

2. Option 2, Allocate additional runtime data sets: You can ignore this step at this point because OMEGAMON II for CICS has no additional data sets allocated in this step.
3. Option 3, Create runtime members: Generates a JOB that can be submitted to create the required configuration members. This Job runs with a return code zero.
4. Option 4, Complete the configuration: Provides a checklist of the steps that you must complete. This describes such tasks as copying the started task JCL to the system proclib libraries, authorizing the required libraries, moving the VTAM definitions to the correct libraries, and installing the components in the CICS regions.

This completes the required steps to configure OMEGAMON II for CICS. The following optional steps are necessary, depending upon the options you want to use and the other products installed in the system.

1. Option 5, Allocate/initialize task history data sets: Is required if you require task history data to be maintained after a CICS region is shutdown.
2. Option 6, Manage CICS global data area(s): Where you can create new or edit existing global data areas. The global specifies the following:
 - The features that are to be active for a CICS region.
 - The group definitions for response time analysis and bottleneck analysis.
 - The rules that are active for the resource limiting feature.
 - The location of the task history data.

When you edit a global data area, there is a comprehensive help system that is available that describes each option in detail. This help is available by pressing PF1. The help is context sensitive and displays information based upon the location of the cursor. Upon exiting the edit, the global data area is validated and any errors are displayed.

After you make the changes the global data areas, copy them to the RKANPARU data sets. Selecting the option to copy global definitions generates JCL to move the members. This JCL specifies DISP=OLD for the RKANPARU data set. If you do not want to shut down all of the tasks that use this library, change this to DISP=OLD. The Job will not run until you exit the configuration tool completely to free up the data sets that are allocated to your TSO ID.

3. Option 7, Modify menu system command security: Use to specify the type of security that is to be active for the menu system commands. You can use it to set passwords for authorized commands, if an external security manager is not to be used. You can also use it to specify the level of security that is required for specific commands.
4. Option 8, Install OMEGAMON subsystem: Only required if this was not done for another OMEGAMON product on this LPAR.

OMEGAMON XE for CICS features

OMEGAMON XE for CICS provides a number of distinct features to allow you to monitor their CICS regions. Although much of the data that is provided shows the current state of resources or the usage of such resources. Some of the features use the collected data to provide real time or near real time summaries of the data that is collected, which provides users with extra insight into the performance of their CICS regions without having to run batch jobs to process large amounts of data.

Resource monitoring

The majority of the workspaces that OMEGAMON XE for CICS provides fall into the category of resource monitoring. They provide information that relates to the current state and usage of the resources that are available to the CICS region, which includes resources that are implicit, such as storage and those that are defined, such as files, programs, and so on.

Resource monitoring primarily allows for the creation of situations that allow the user to be notified when resources are unavailable, resource consumption exceeded certain thresholds, or available resources are below safe values. Careful tuning of the situations on a system can help to ensure that support staff are notified before the system availability is threatened.

OMEGAMON CICS currently provides information relating to the following resources in a CICS region:

- ▶ Automatic Initiate Descriptors (AIDs)
- ▶ Business Transaction Services process type details
- ▶ Business Transaction Services process details
- ▶ Business Transaction Services containers
- ▶ MRO connections
- ▶ ISC connections
- ▶ IP connections
- ▶ DB2 subsystem
- ▶ DB2 transactions
- ▶ DB2 thread activity
- ▶ Database control for IMS
- ▶ Dispatcher Summary
- ▶ Dispatcher TCB pools
- ▶ Dispatcher TCB modes
- ▶ Dump details
- ▶ Enqueue contention
- ▶ Enterprise Java Corba servers
- ▶ Enterprise Java request models
- ▶ Enterprise Java deployed java programs (DJARs)
- ▶ Exit programs
- ▶ File control data sets
- ▶ CICS region data sets
- ▶ Interval control elements
- ▶ Java programs
- ▶ CICS Journals
- ▶ CICS JVMs

- ▶ JVM profiles
- ▶ JVM classcaches
- ▶ Log streams
- ▶ LSR pools
- ▶ MQ status
- ▶ MVS TCBs
- ▶ Recovery manager Unit of work links
- ▶ Storage usage
- ▶ Storage DSA usage
- ▶ Storage Subpool usage
- ▶ System initialization values
- ▶ Transaction classes
- ▶ TCP/IP activity
- ▶ TCP/IP services
- ▶ Temporary storage
- ▶ Temporary storage queues
- ▶ Terminal storage violations
- ▶ Transactions
- ▶ Transient data
- ▶ Transient data queues
- ▶ Transaction manager
- ▶ UOW disposition
- ▶ UOW enqueues
- ▶ VSAM files
- ▶ VSAM RLS conflicts
- ▶ Web services
- ▶ Web services virtual hosts
- ▶ Web services pipelines
- ▶ Workrequests

In addition to this list, there is the Region Overview query and workspace that provides information from several places in one query to allow for many important metrics to be returned in a single query. Region overview provides information, such as the transaction rate, the percent of maximum tasks, the I/O rate, CPU rate, and other details.

Service level analysis

Service level analysis is a great way to determine if transactions are meeting their performance objectives. It provides details relating to whether transactions are meeting their performance objectives. The objectives can be based upon the average response time or by percent of transactions that meet their goal response time.

Service level analysis produces summaries at the LPAR level. You can get the report selecting the CICS group node for an LPAR on the physical tree, as shown in Figure 10-33.

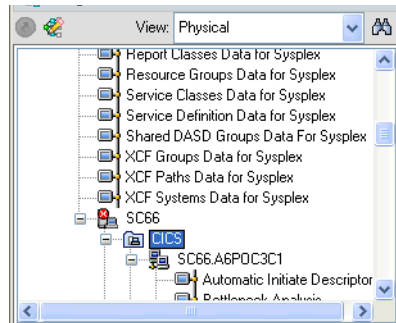
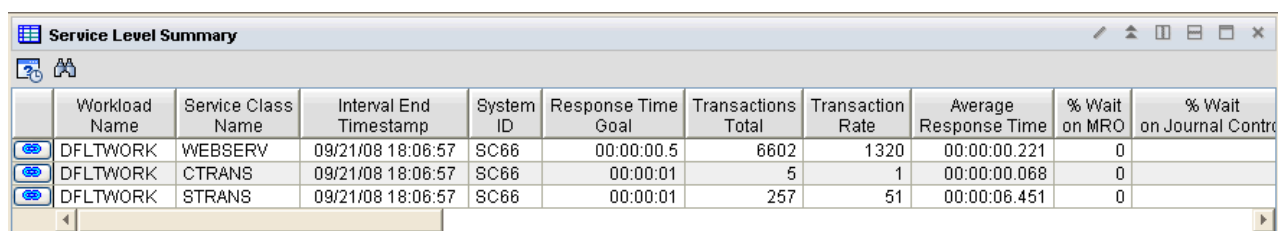


Figure 10-33 CICS Group node

The Service Level Summary report appears on the bottom pane of the workspace.



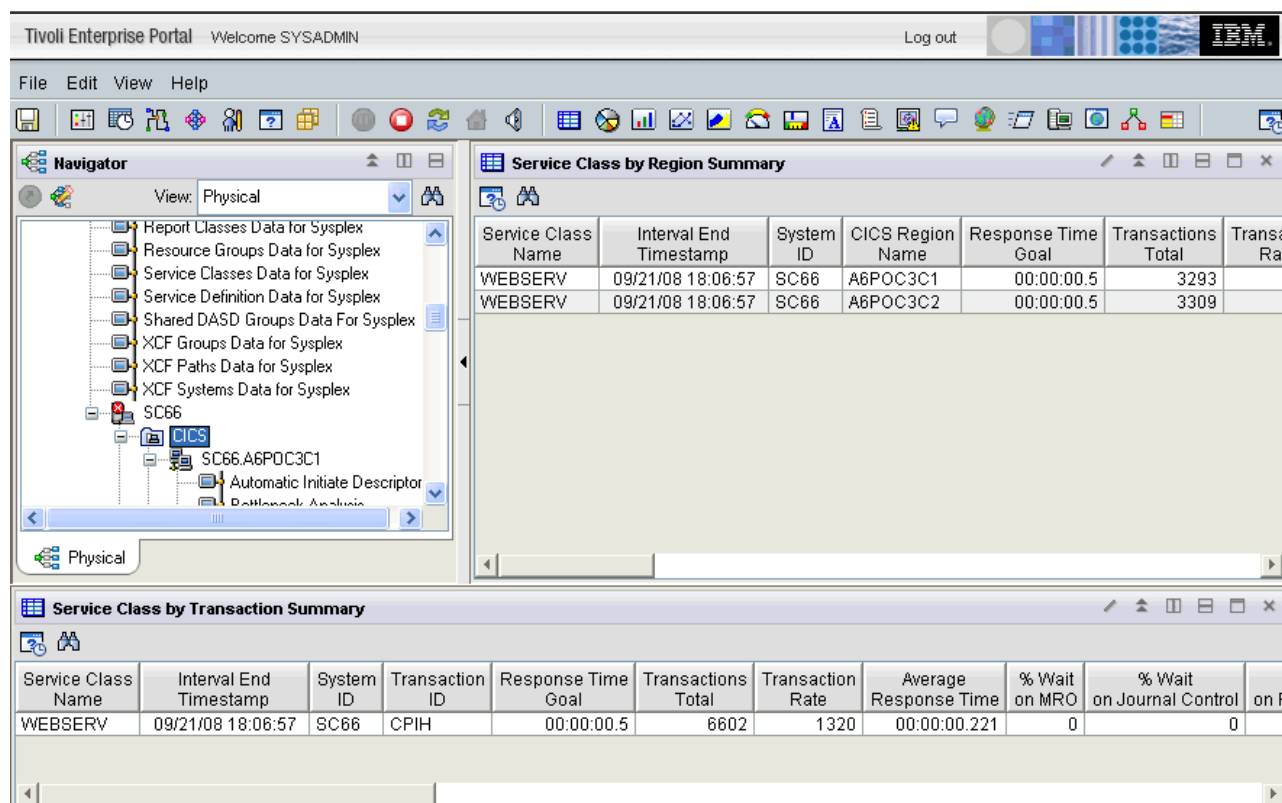
The image shows a window titled "Service Level Summary" with a table of transaction data. The table has columns for Workload Name, Service Class Name, Interval End Timestamp, System ID, Response Time Goal, Transactions Total, Transaction Rate, Average Response Time, % Wait on MRO, and % Wait on Journal Control. There are three rows of data for DFLTWORK workload.

Workload Name	Service Class Name	Interval End Timestamp	System ID	Response Time Goal	Transactions Total	Transaction Rate	Average Response Time	% Wait on MRO	% Wait on Journal Control
DFLTWORK	WEBSERV	09/21/08 18:06:57	SC66	00:00:00.5	6602	1320	00:00:00.221	0	
DFLTWORK	CTTRANS	09/21/08 18:06:57	SC66	00:00:01	5	1	00:00:00.068	0	
DFLTWORK	STRANS	09/21/08 18:06:57	SC66	00:00:01	257	51	00:00:06.451	0	

Figure 10-34 Service Level Summary report

The Service Level Summary report shows the summary for all of the transactions in all of the CICS regions for an LPAR that match the service Level analysis definition.

Selecting the link to Service Class Summary produces the following workspace in Figure 10-35.



The image shows the Tivoli Enterprise Portal workspace. On the left is a Navigator pane showing a tree view of system components, with "CICS" and "SC66.A6POC3C1" selected. The main area contains two reports. The top report is "Service Class by Region Summary" and the bottom report is "Service Class by Transaction Summary".

Service Class by Region Summary

Service Class Name	Interval End Timestamp	System ID	CICS Region Name	Response Time Goal	Transactions Total	Transaction Rate
WEBSERV	09/21/08 18:06:57	SC66	A6POC3C1	00:00:00.5	3293	
WEBSERV	09/21/08 18:06:57	SC66	A6POC3C2	00:00:00.5	3309	

Service Class by Transaction Summary

Service Class Name	Interval End Timestamp	System ID	Transaction ID	Response Time Goal	Transactions Total	Transaction Rate	Average Response Time	% Wait on MRO	% Wait on Journal Control	on F
WEBSERV	09/21/08 18:06:57	SC66	CPIH	00:00:00.5	6602	1320	00:00:00.221	0	0	

Figure 10-35 Service Class Summary workspace

The Service Class Summary workspace shows two reports:

- ▶ To the right of the window is the Service Class by Region Summary, which provides a summary of all the CICS regions where any tasks that were classified in the service class run.
- ▶ The lower portion of the window is the Service Class by Transaction, which shows all the transaction IDs that run that were classified in this service class. In this case, there is only one, CPIH, but there is no restriction on the number of transactions that can be part of a service class.

The classification rules used by OMEGAMON XE for CICS to produce the summaries can either be those defined to z/OS Workload Manager or defined in OMEGAMON XE for CICS.

Configuring Service Level Analysis

To configure OMEGAMON XE for CICS you must first enable the CICS SLA view by adding the CICS SLA view to the assigned views for your user ID:

1. Click the **Administer Users** icon, Figure 10-36, in the top-left corner of the TEP display.

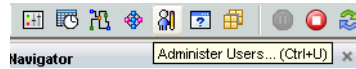


Figure 10-36 Administer Users icon

2. Select your current USERID and the Navigator Views tab, as shown in Figure 10-37.
3. Ensure that CICS SLA is in the Assigned View box. If it is not, in the available views box, click **CICS SLA**, and then click the left arrow.

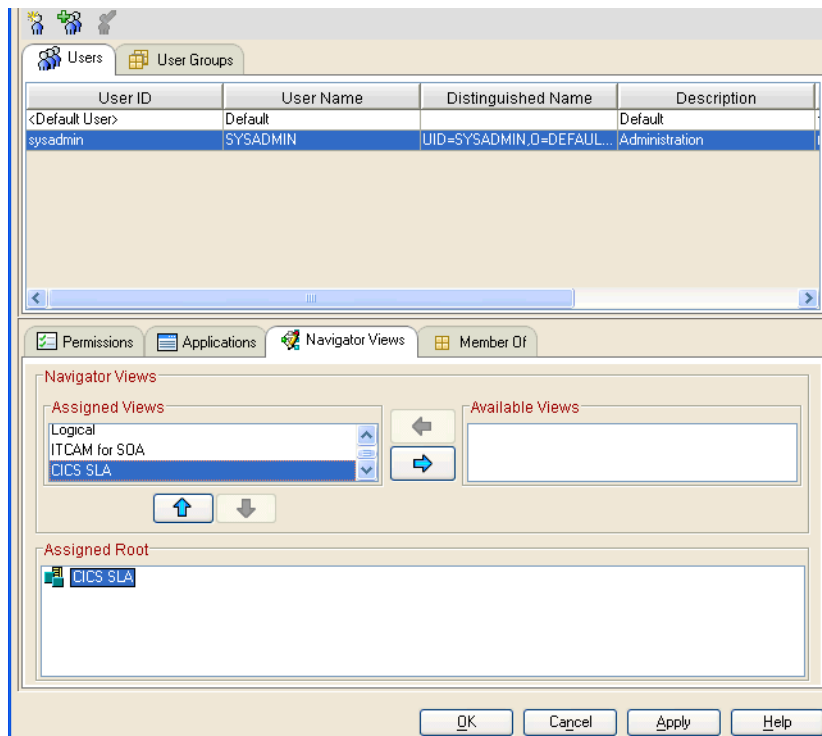


Figure 10-37 Administer Users Navigator Views

4. After the CICS SLA view is added to the user, you can select it from the navigator pane in the TEP, as shown in Figure 10-38 on page 270.

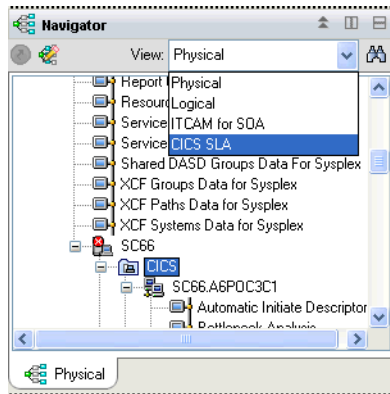


Figure 10-38 Selecting CICS SLA view

Figure 10-39 shows the CICS SLA view.

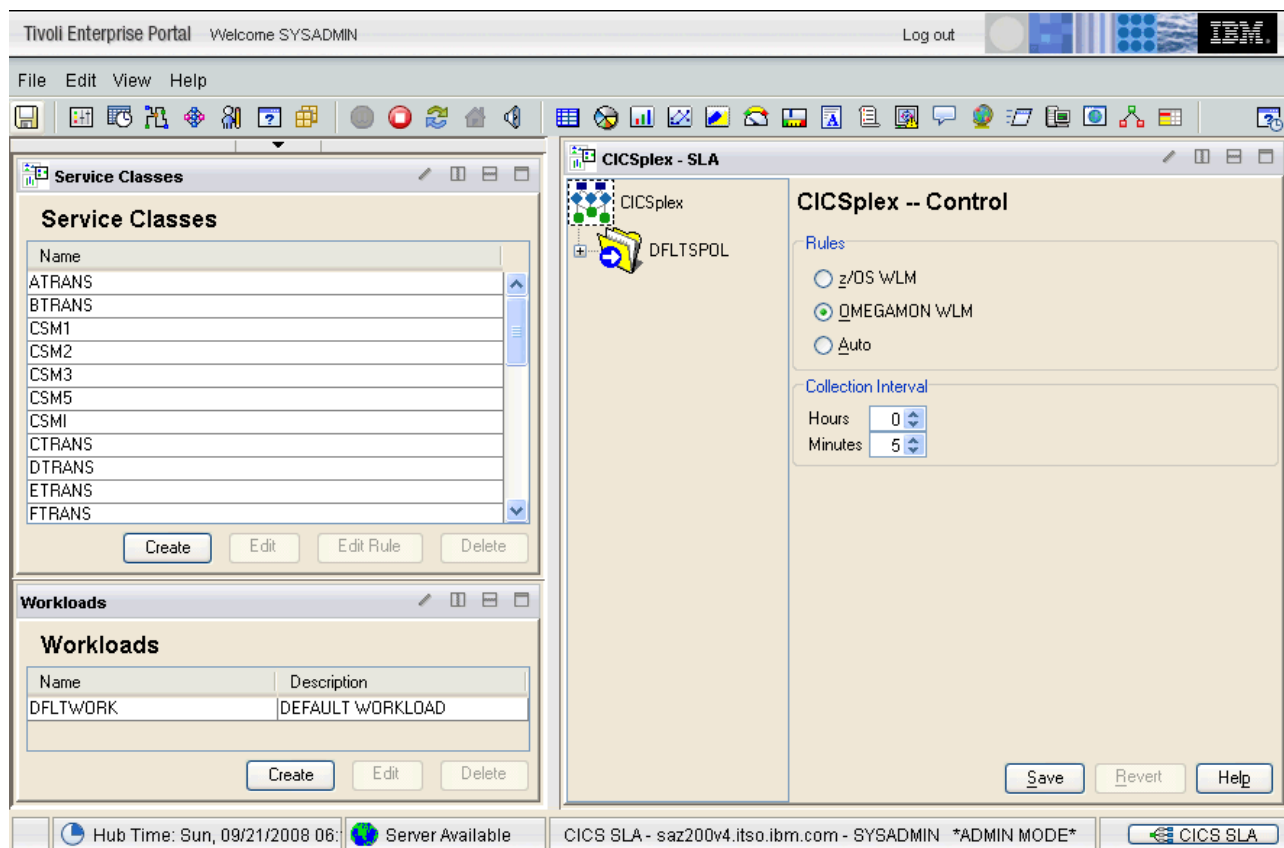


Figure 10-39 CICS SLA view

To the right of the view we see the CICSplex Control information and the default Service Policy of DFLTSPOL.

In the CICSplex Control we configure how the classification is to take place:

- ▶ **z/OS WLM:** Indicates that the WLM assigned service class name will be used to classify the transactions.
- ▶ **OMEGAMON WLM:** Indicates that the rules that are defined using this interface determine the service class names that the transactions are classified in.
- ▶ **Both:** Indicates that the z/OS WLM classification will be used, if available. If the service class cannot be determined this way, the OMEGAMON rules classify the task.

The collection interval determines how often response data for service classes, CICS regions, and individual transactions is summarized and reported through the Service Class Analysis workspace.

A service policy applies to all service classes that are defined for your enterprise. Service-class goals vary by service policy: Service policies let you override a service class's response-time goals as dictated by your site's varying requirements, for example, you can have one service policy for prime-shift operation, another for nighttime operation, and a third for weekend operation.

To the left of the view we have service classes and workload groups. A workload group is one or more service classes that you to monitor as a unit. With it, you can monitor related service classes and highlight the worst-performing class within the group.

A service class identifies a block of related transactions that share common response-time goals for a single workload. The transactions must be related by transaction name, the user ID that invoked them, the VTAM terminal ID (LU name) that submitted them, the CICS region running them, or any combination thereof.

To define a new Service Class, Figure 10-40:

1. In the Service Class pane, click **Create**.

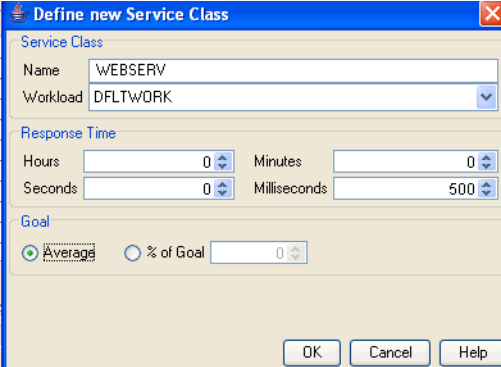


Figure 10-40 Define new Service Class window

2. Within the service-class editor, define a name for your new service class. The name you supply is converted to uppercase. Pull down the list of available workloads, and select the existing workload that you want to associate this service class with.
3. Within the Response Time pane, specify the response time that you expect for the CICS transactions that are associated with this service class.

4. Within the Goal pane, select one of these options:
 - Activating the Average radio button means that the average response time for all matching transactions must at least meet the Response Time specified % of Goal
 - Activating the % of Goal radio button and specifying a percentage mean that the given percentage of matching transactions must at least meet the Response Time specified
5. Press OK to accept the values.
6. Next you must create rules that control which transactions are classified as a part of this service class. Highlight the service class, and press **Edit Rule**. Figure 10-41 shows the rule definition window.

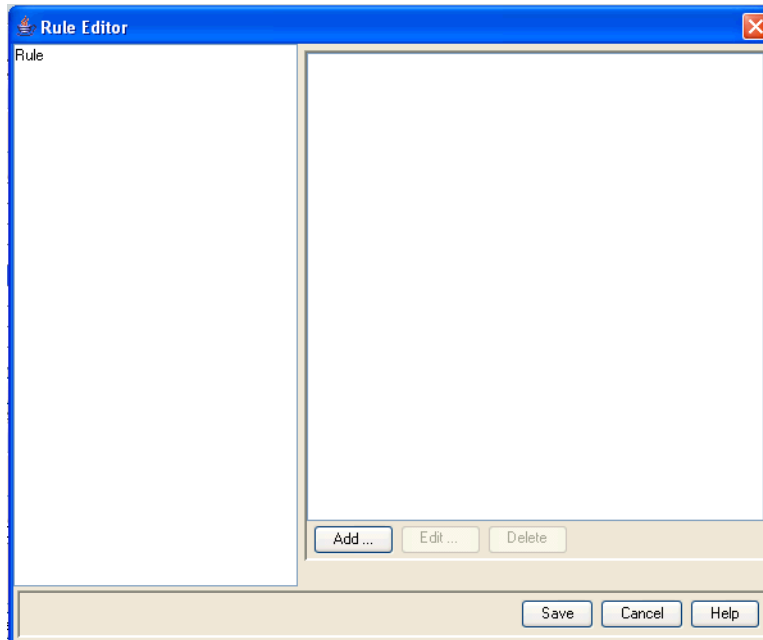


Figure 10-41 Rule definition window

7. Right-click a rule in the left pane, and a submenu is displayed, as shown in Figure 10-42.

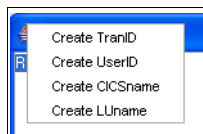


Figure 10-42 Rule pop-up menu

8. Select either:
 - Create TranID, to select based upon Transaction ID.
 - Create UserID, to select based upon the user ID that is associated with the transaction.
 - Create CICSName, to select based upon the JOB name of the CICS region.
 - Create LUName, to select based upon the VTAM LU name for the terminal where the transaction originated.

A pop-up is then displayed where you can enter a value, as shown in Figure 10-43 on page 273.

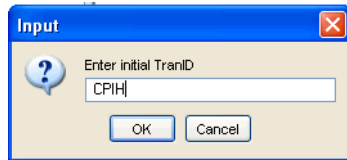


Figure 10-43 pop-up window to enter rule value

9. Repeat these steps for other values, as shown in Figure 10-44.

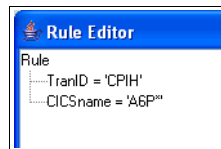


Figure 10-44 completed rules

In this case, all transactions that are running in CICS regions, with job names starting with the characters *A6P* and having the transaction ID of *CPIH*, are classified in this service class.

Bottleneck Analysis

Bottleneck Analysis is a useful tool for people who want to improve the performance of the applications that run on a CICS region. Getting the most performance out of a CICS application can be compared to tuning almost anything, including a car! Extracting the maximum performance involves maximizing the time that is spent on productive work and minimizing that which puts a strain on the system.

In computer software terms, this means to minimize the time that an application must wait for something. If an application spends the bulk of its time waiting for a resource, such as storage, or a LSR buffers, then the application is not providing the maximum performance. In addition to elongated response times, the transactions hold resources themselves for longer and therefore potentially add to the contention occurring in the system.

Bottleneck Analysis helps you identify the wait reasons that your applications are experiencing. Every task in the system has a wait reason identified. That wait reason might be Running, in which case the task is doing productive work. Bottleneck Analysis has a list of wait reasons that it understands. Bottleneck Analysis runs as a subtask of the KOCCI. It periodically scans all of the tasks in the CICS region and accumulates counts for each of the wait reasons. Wait reasons that it knows nothing about are accumulated in an UNKNOWN bucket. If a task is in a wait reason that is turned off in the Bottleneck Analysis table, it is not counted.

By carefully selecting which wait reasons are active for a given system, it is possible to see those wait reasons that are truly impacting your applications and therefore what resources your applications are waiting for.

Figure 10-45 on page 274 shows an example of a Bottleneck Analysis.

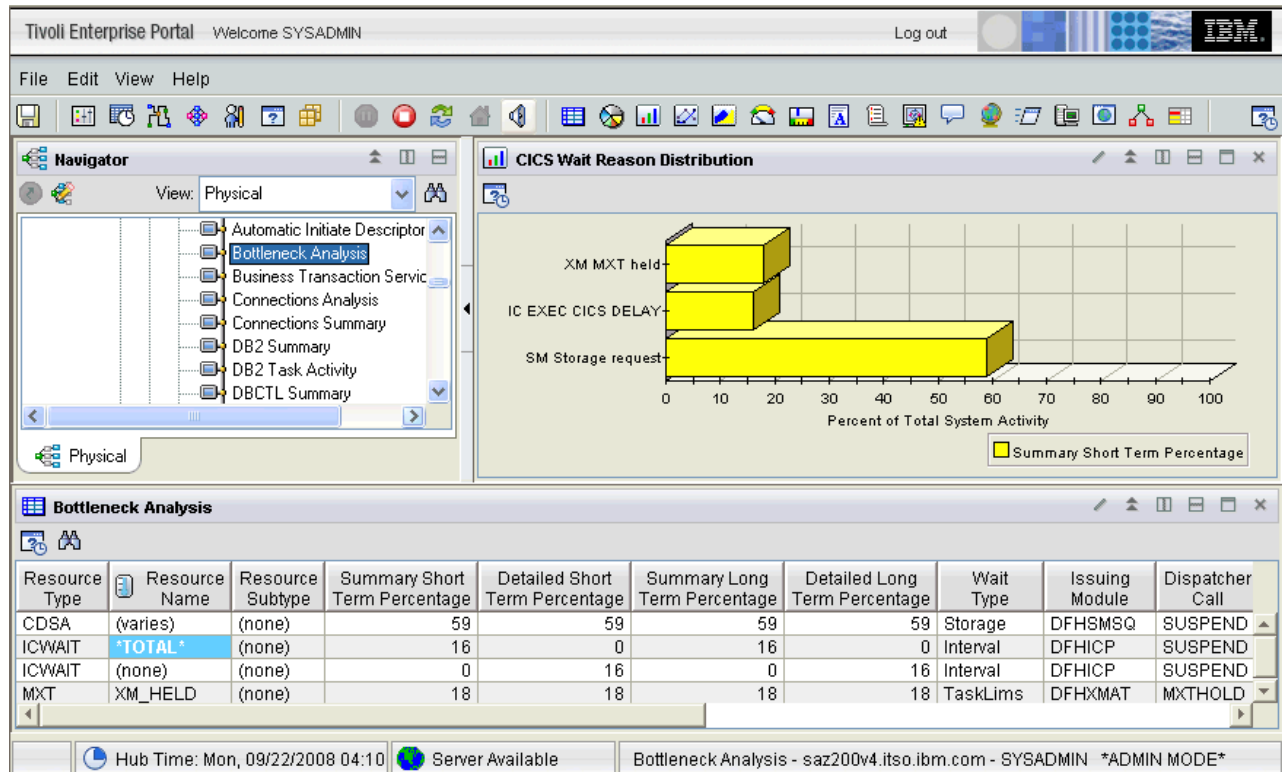


Figure 10-45 Bottleneck Analysis example

In the example in Figure 10-45, there are three wait reasons that were identified:

- ▶ **IC EXEC CICS DELAY:** An application issue because it indicates that the task is explicitly waiting for a given period of time. There is not much from a performance stand point that you can do to improve this.
- ▶ **XM MXT HELD:** Indicates that 18 percent of the recorded wait reasons were for tasks that were delayed because of MAXTASKS. This might be because the value is too low.
- ▶ **SM STORAGE REQUEST:** Indicates that nearly 60 percent of the samples show that transactions are waiting for storage requests to be honored, which is the largest wait reason, so examine it first. It might be that the DSALIMIT is too low or that the application will simply not fit in one CICS region and some of the workload needs to be spread across other regions.

The same information can be displayed in the menu system, as seen in Figure 10-46 on page 275.

```

=====
ZBPDEX      VTM      A6POC3C1 V560./C SC66 09/22/08 16:15:53
> PF1 Help   PF3 Back   PF4 Main Menu   PF7 Up   PF8 Down   PF11 Zoom

> A-Bottleneck Graph   B-Bottlenecks   C-Group Graph   D-Group Bottlenecks
> E-Impact Analysis    F-Impact Profile   G-Impact Detail   H-Enqueues
=====
>
      BOTTLENECKS

PDEX
+
+ Resource      Resource
+ Type          Name          Short Term Information          Long Term Information
+ -----
+ % 0          50          100          % 0          50          100
+ CDSA          (varies)      59 |-----> . . |      59 |-----> . . |
+ ICWAIT        *TOTAL*      16 |--> . . . . |      16 |--> . . . . |
+              (none)      (16) |--> . . . . |      (16) |--> . . . . |
+ MXT           XM_HELD      18 |--> . . . . |      18 |--> . . . . |
+
+              Samples . . . : 43318      Samples . . . : 43318
+              Elapsed . . . : 5:59 MN      Elapsed . . . : 5:59 MN
+              Interval . . . : 10:00 MN      Interval . . . : 30:00 MN
=====

```

Figure 10-46 Bottleneck Analysis in the Menu system

Controlling Bottleneck Analysis

Bottleneck Analysis is controlled through the global data area. The enablement of the feature is controlled in the <STARTUP_CONTROL> section, as shown in Example 10-1.

Example 10-1 Enabling Bottleneck Analysis

```

*
  <STARTUP_CONTROL>
*
  BOTTLENECK_ANALYSIS=AUTO

The BOTTLENECK_OPTIONS section specifies the parameters for the subtask:

*
  <BOTTLENECK_OPTIONS>
*
  CLEAR_INTERVAL_LONG=30
  CLEAR_INTERVAL_SHORT=10
  SAMPLE_INTERVAL=20
  VARIABLE_BUCKETS=1000
  EXCLUDED_TRANS=(CSSY,CSJC,CVST,CSSX,CSGX,CSNC,DSNC,CFQ*,KD4*)

```

Notes about the <STARTUP_CONTROL> section of Example 10-1:

- ▶ The *CLEAR_INTERVAL_LONG* and *CLEAR_INTERVAL_SHORT* controls the time span for the long and short term accumulations. Providing long and short term displays makes it possible to determine if a particular wait reason is a sudden change in the profile of an application or a longer term trend.
- ▶ The *SAMPLE_INTERVAL* value specifies, in tenths of a second, how often the subtask samples the active CICS region. Bottleneck Analysis can be a heavy user of CPU. The interval by default is set to two seconds. However in a busy system, this might be too high. Try to increase this value while still attempting to provide a statistically valid sample. It serves no purpose to make the interval such that only a few samples are accumulated for each wait reason.

- ▶ The *VARIABLE_BUCKETS* value specifies how many slots are reserved to Wait reasons that have a variable portion to their name. Bottleneck Analysis does not expand the slots that are available, so if a value of 1000 is specified, and more variable wait reasons than this are encountered, some information is lost. The default here is normally sufficient.
- ▶ The *EXCLUDED_TRANS* keyword specifies those transactions that are to be ignored by Bottleneck Analysis. It serves no purpose to collect information about system tasks and background server transaction that are permanently active, unless that is the workload you are interested in tuning.
- ▶ The last item of control is the list of wait reasons, Figure 10-47, which you can see by using the menu system option O.J.

ZCDLST VTM CIWSS3C2 V560./C SC66 09/22/08 17:03:39						
> PF1 Help	PF3 Back	PF4 Main Menu	PF7 Up	PF8 Down	PF11 Zoom	
> A-RTA On	B-RTA Off	C-RTA Status	D-RTA Intervals	E-RTA Scaling		
> F-ONDV On	G-ONDV Off	H-ONDV Status	I-Bottleneck Ctl	J-Wait Reasons		
> K-INTR Ctl	L-IANL On	M-IANL Off	N-IANL Settings	O-IANL Groups		
> P-Collection	Q-Shutdown	R-RLIM On	S-RLIM Off	T-RLIM Status		
> U-SMF Status	V-ATF Filters	W-ATF Status				
=====						
>	CONTROL BOTTLENECK ANALYSIS WAIT REASON BUCKETS					
BLST						
+ BLST	On/	Resource	Resource	Issuing	Wait Reason	Wait
+ ID	Off	Type	Name	Module	Description	Type
+ ----						
: SY1W	OFF	(none)	(none)	DFHDUIO	DU: Dump dataset I/O	Systasks
: SY2W	OFF	(none)	(none)	DFHTISR	TI: Timer service rq	Systasks
: RMSL	ON	(none)	(none)	DFHRMSL7	RM: Keypoint process	Systasks
: ZNAC	ON	(none)	(none)	DFHZNAC	ZC: Terminal error	Systasks
: DLCN	ON	(none)	DLCNTRL	DFHDBCT	DBCTL: Work element	DBCntl

Figure 10-47 Menu system wait reason control

Certain wait reasons are turned off for Bottleneck Analysis monitoring by default. If you determine that a wait reason is active and that you want to disable it, you can temporarily achieve this by changing its On/Off value or on a more permanent basis by coding the *BOTTLENECK_ANALYSIS* section in the global as shown in Example 10-2.

Example 10-2 Coding the *BOTTLENECK_ANALYSIS* section

```
*
<BOTTLENECK_ANALYSIS>
*
DSDF=NO
SODM=NO
SMRE=NO
```

Transaction History

It is often very useful to look back at recent transactions to see, in detail, the response time for a particular transaction or possibly look at resource consumption for an individual task. Although CICS SMF data provides the detailed information for a transaction, it can be a cumbersome task to gain access to recent SMF data and run an ad hoc report.

OMEGAMON XE for CICS provides a feature called Transaction History, also known as Online Data Viewing (ONDV). This feature creates a data store for each CICS region and manages the space to hold as much transaction data as possible. Transaction History stores comprehensive information about a transaction with detailed file and database requests are collected.

You can select Transaction History by using the Online Data Viewing workspace under Transactions Analysis for a CICS region in OMEGAMON XE for CICS, as shown in Figure 10-48.

System ID	CICS Region Name	CICS Version	End Time	Transaction ID	Task Number	Terminal ID	Transaction Type	User ID	Program ID	CPU Time
SC66	CIWSS3C2	6.5.0	09/22/08 18:11:39	CSOL	00004	n/a	TRM	CIWS3D	DFHSOL	00:00:00
SC66	CIWSS3C2	6.5.0	09/22/08 17:40:11	CSOL	00004	n/a	TRM	CIWS3D	DFHSOL	00:00:00
SC66	CIWSS3C2	6.5.0	09/22/08 17:37:20	CWBG	79601	n/a	SYS	CIWS3D	DFHWBGB	00:00:00
SC66	CIWSS3C2	6.5.0	09/22/08 17:08:44	CSOL	00004	n/a	TRM	CIWS3D	DFHSOL	00:00:00.35
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDR	78882	n/a	TRM	PRIVAT01	DFHPIDSH	00:00:00.01
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDR	78877	n/a	TRM	PRIVAT01	DFHPIDSH	00:00:00.01
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDR	78881	n/a	TRM	PRIVAT01	DFHPIDSH	00:00:00.01
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDS	79289	n/a	TRM	USERWS01	DFHPIAP	00:00:00
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDS	79288	n/a	TRM	USERWS01	DFHPIAP	00:00:00
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDR	78873	n/a	TRM	PRIVAT01	DFHPIDSH	00:00:00.01
SC66	CIWSS3C2	6.5.0	09/22/08 16:45:12	ORDS	79287	n/a	TRM	USERWS01	DFHPIAP	00:00:00

Figure 10-48 Transaction history display in TEP

The Menu system provides an equivalent display, as shown in Figure 10-49.

```

      ZONDV   VTM   A6POC3C1 V560./C SC66 09/22/08 18:40:02
> PF1 Help   PF3 Back   PF4 Main Menu   PF7 Up   PF8 Down   PF11 Zoom

>           A-History Record View           B-History Record Selection
>           C-Trace Record View             D-Trace Filters Management
=====
>           HISTORICAL TRANSACTION OVERVIEW
>
ONNDV
+           Transaction Overview: 09/22/08
+
+   End   Tran   Task   Term   Type   CPU   Resp   Storage   File   Term   Abend
+   Time   ID     Num    ID     Type   Time   Time    HMM     Reqs   I/O    Code
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+ 16:03:09 CPIH  15713  n/a   TRM    .2 2.046357 33335K  3 0
+ 16:03:09 CPIH  15706  n/a   TRM    .2 1.980787 33335K  3 0
+ 16:03:09 CPIH  15711  n/a   TRM    .2 1.828366 33335K  3 0
+ 16:03:09 CPIH  15721  n/a   TRM    .2 1.544157 33335K  3 0
+ 16:03:09 CPIH  15720  n/a   TRM    .2 1.513379 33335K  3 0
+ 16:03:08 CPIH  15704  n/a   TRM    .2 1.558967 33335K  3 0
+ 16:03:08 CPIH  15699  n/a   TRM    .2 1.588833 33335K  3 0
+ 16:03:08 CPIH  15696  n/a   TRM    .2 2.098515 33335K  3 0
+ 16:03:08 CPIH  15695  n/a   TRM    .2 2.027101 33335K  3 0
+ 16:03:08 CPIH  15694  n/a   TRM    .2 2.195676 33335K  3 0
+ 16:03:08 CPIH  15679  n/a   TRM    .2 2.673689 33335K  3 0
+ 16:03:08 CPIH  15689  n/a   TRM    .2 2.423562 33335K  3 0
+ 16:03:08 CPIH  15690  n/a   TRM    .2 2.392998 33335K  3 0
+ 16:03:08 CPIH  15684  n/a   TRM    .2 2.577388 33335K  3 0
+ 16:03:08 CPIH  15686  n/a   TRM    .2 2.372017 33335K  3 0

```

Figure 10-49 Transaction History in the Menu system

The menu system also allows for the display of comprehensive detail information relating to a transaction. Selecting one of the tasks in the display in Figure 10-49 using the zoom key (PF11) provides the display in Figure 10-50 on page 278 and Figure 10-51 on page 279.

```

ZZONDV   VTM   A6POC3C1 V560./C SC66 09/22/08 18:43:48
> PF1 Help   PF3 Back   PF4 Main Menu   PF7 Up   PF8 Down

=====
>                                HISTORICAL TRANSACTION DETAIL
ONDV 08
+
+                                Task Detail Information
+
+                                General Information
+ Transaction ID . . . . . : CPIH      Task number . . . . . : 15696
+ Userid . . . . . : CICSUSER    Luname . . . . . : None
+ Facility ID (local) . . : n/a     Facility type (local) . : Term
+ Real transaction ID . . : CPIH     Umbrella transaction ID : None
+ Program ID - first . . : DFHPIDSH Umbrella program . . . : None
+ Abend code . . . . . : None
+
+                                Time Statistics
+ CPU time . . . . . : 0.173600    Overall elapsed time . . : 2.098515
+ Dispatch time . . . . . : 0.267648 Total wait time . . . . : 1.830848
+ Re-dispatch wait time . . : 0.132016 Exception wait time . . : 0.000000
+ TS VSAM I/O wait time . . : 0.000000 TD VSAM I/O wait time . . : 0.000000
+ File I/O wait time . . . : 0.000000 JC I/O wait time . . . : 0.000000
+ TC I/O wait time . . . . : 0.000000 MRO wait time . . . . . : 0.000000
+ 1st dispatch delay time : 0.000112 Transaction class delay : 0.000000
+ Max tasks delay . . . . . : 0.000000 Local ENQ delay . . . . : 0.000000
+ LU61 wait time . . . . . : 0.000000 LU62 wait time . . . . . : 0.000000
+ FEPI wait time . . . . . : 0.000000 RMI elapsed time . . . . : 0.000016
+ RMI suspend time . . . . : 0.000000 RLS file I/O wait time : 0.000000
+ Syncpoint elapsed time : 0.000656 Lock manager delay . . . : 1.288544
+ WAIT EXTERNAL wait time : 0.000000 WAITCICS and WAIT EVENT : 0.000000
+ Interval control wait . . : 0.000000 Dispatchable wait time : 0.000000
+ Shared TS I/O wait time : 0.000000 RLS CPU time . . . . . : 0.000000
+ IMS wait time . . . . . : 0.000000 DB2 Readyq wait time . . : 0.000000
+ DB2 Connection wait time : 0.000000 DB2 wait time . . . . . : 0.000000
+ SOCKET I/O wait time . . : 0.505376 Global ENQ delay . . . . : 0.000000
+ RRMS/MVS wait time . . . : 0.000000 MAXOPENTCBS delay time : 0.000000
+ JVM elapsed time . . . . . : 0.000000 JVM suspend time . . . . : 0.000000
+ QR TCB wait-for-dispatch : 0.012592 QR TCB elapsed time . . . : 0.017008
+ QR TCB CPU time . . . . . : 0.000720 Other TCBs elapsed time : 0.006112
+ Other TCBs CPU time . . . : 0.005984 JVM(J8) TCB CPU time . . : 0.000000
+ LE(L8) TCB CPU time . . . : 0.166880 SS(S8) TCB CPU time . . . : 0.000000
+ Program fetches wait . . : 0.000000 Wait for a JVM TCB . . . : 0.000000
+ JVM initialisation time : 0.000000 JVM reset time . . . . . : 0.000000
+ Key 8 TCB elapsed time : 0.244512 Key 8 TCB CPU time . . . : 0.166880
+ Key 9 TCB elapsed time : 0.000000 Key 9 TCB CPU time . . . : 0.000000
+ J9 TCB CPU time . . . . . : 0.000000 Wait for H8 TCB time . . : 0.000000
+ R0 TCB elapsed time . . . : 0.000000 R0 TCB CPU time . . . . . : 0.000000
+ TCB mismatch time . . . . : 0.000000 TCB change mode delay . . : 0.036784
+ TCB create delay . . . . . : 0.000000 3270 Partner wait time : 0.000000
+
+                                General Statistics
+ Primary term input msgs : 0      Primary term output msgs: 0
+ Primary term input chars: 0      Primary term output chars: 0
+ Sec LU61 input msgs . . . : 0     Sec LU61 output msgs . . : 0
+ Sec LU61 input chars . . : 0     Sec LU61 output chars . . : 0

```

Figure 10-50 Menu system Task History detail


```

+   Sec LU62 input msgs . . : 0   Sec LU62 output msgs . . : 0
+   Sec LU62 input chars . . : 0   Sec LU62 output chars . . : 0
+   TD gets . . . . . : 0   TD puts . . . . . : 0
+   TD purges . . . . . : 0   TS gets . . . . . : 0
+   TS puts to aux . . . . : 0   TS puts to main . . . . : 0
+   Shared TS wait count . . : 0   PC links . . . . . : 11
+   PC loads . . . . . : 0   PC xctls . . . . . : 0
+   JC writes . . . . . : 0   IC starts . . . . . : 0
+   Syncpoint requests . . . : 1   BMS requests . . . . . : 0
+   BMS map requests . . . . : 0   BMS in requests . . . . : 0
+   BMS out requests . . . . : 0   CICS logger writes . . . : 0
+   PC link URMs . . . . . : 0   IC requests . . . . . : 0
+   3270 bridge tran ID . . . : n/a TS total requests . . . . : 0
+   DPL requests . . . . . : 0   IMS/DBCTL requests . . . : 0
+   DB2 requests . . . . . : 0   OO class requests . . . . : 0
+   SSL bytes encrypted . . . : 0   SSL bytes decrypted . . . : 0
+   CICS TCBS attached . . . : 0   TCB Mode Switches . . . . : 110
+   WEB Receive requests . . . : 1   WEB Characters received : 0
+   WEB Send requests . . . . : 0   WEB Characters sent . . . : 0
+   WEB total request count : 12   WEB Repository READs . . : 0
+   WEB Repository WRITES . . : 0   Local container bytes . . : 0
+   Remote IC channel starts: 0   Remote IC channel data . . : 0
+   Total channel requests : 172   Browse channel requests : 0
+   Get container requests : 80   Put container requests : 88
+   Move container requests : 4   Total bytes for GETs . . : 36710198
+   Total bytes for PUTs . . : 31465325 DPL CHANNEL data bytes : 0
+   DPL RETURN CHNL bytes . . : 0   LINK with CHANNEL . . . . : 2
+   XCTL with CHANNEL . . . . : 0   DPL with CHANNEL . . . . : 0
+   RETURN with CHANNEL . . . : 0   RETURN CHNL bytes . . . . : 0
+   Client IP address . . . . : 10.1.100.43
+   Tran Group ID (char) . . . : ..USIBMSC.A6POC3C1C....zc..
+   Tran Group ID (hex) . . . : 11EECCDEC4CFDDCF3C0126AA800
+   904292423B1676333139DD8F9300
+   TRGID: X'1910E4E2C9C2D4E2C34BC1F6D7D6C3F3C3F1C3091D2D68AFA9830000'
+
+                               Storage Statistics
+   Getmains <16M . . . . . : 0   Getmains >16M . . . . . : 56
+   HWM <16M . . . . . : 0   HWM >16M . . . . . : 32553K
+   Occupancy <16M . . . . . : OK Occupancy >16M . . . . . : 7570M
+   HWM of total pgm storage: 70K
+   HWM of pgm storage <16M : 0   HWM of pgm storage >16M : 70K
+   HWM pgm storage cdsa . . . : 0   HWM pgm storage ecdsa . . : 0
+   HWM pgm storage rdsa . . . : 0   HWM pgm storage erdsa . . : 40K
+   HWM pgm storage sdsa . . . : 0   HWM pgm storage esdsa . . : 30K
+
+                               Application Trace Active
+
+                               File Control Statistics
+   Local Browsers . . . . . : 0   Local Gets . . . . . : 3
+   Local Adds . . . . . : 0   Local Puts . . . . . : 0
+   Local Deletes . . . . . : 0   Total Local Requests . . : 3
+   Local VSAM calls . . . . : 3   Total Remote Requests . . : 0
+   Total Requests . . . . . : 3
+
+                               Umbrella Data
+   User work area (char) . . : n/a
+                               Unit-of-work Information
+   Netname . . . . . : USIBMSC.A6POC3C1....
+   CICS token info (char) : ....n...
+   CICS token info (hex) . . : 012B9100
+   9DD35001

```

Figure 10-51 Menu system Task History detail continued

As you can see from Figure 10-50 on page 278 and Figure 10-51, a large amount of data is available for each task. Additionally, when the File Control Statistics heading is highlighted, it indicates that it is possible to zoom for more details.

Figure 10-52 on page 280 shows the file summary for a task.

```

ZZONDVD VTM A6POC3C1 V560./C SC66 09/22/08 18:57:03
> PF1 Help PF3 Back PF4 Main Menu PF7 Up PF8 Down PF11 ZOOM

=====
> HISTORICAL FILE SUMMARY FOR SELECTED TASK

ONDV 08 FILE SUMMARY
+ Transaction Detail for CPIH task number =15696
+
+
+ File Control Statistics
+
+ Read . . . . . : 3 Read time . . . . . : 0.002944
+ Write . . . . . : 0 Write time . . . . . : 0.000000
+ Update . . . . . : 0 Update time . . . . . : 0.000000
+ Delete . . . . . : 0 Delete time . . . . . : 0.000000
+ Browse . . . . . : 0 Browse time . . . . . : 0.000000
+ Misc request . . . . . : 0 Misc request time . . . : 0.000000
+ Total requests . . . . : 3
+
+ Database Requests Elapsed
+ Time
+ -----
+ EXMPCONF 2 0.001792
+ EXMPCAT 1 0.001152

```

Figure 10-52 File summary for a task

Figure 10-53 shows that the transaction accessed two VSAM files. Again, you can get more detail on each file.

```

ZZONDVD VTM A6POC3C1 V560./C SC66 09/22/08 19:01:03
> PF1 Help PF3 Back PF4 Main Menu PF7 Up PF8 Down

=====
> HISTORICAL FILE DETAIL FOR SELECTED TASK

ONDV 08 FILE DETAIL EXMPCONF
+ Transaction Detail for CPIH task number =15696
+
+
+ File Control Statistics
+
+ Database . . . . . : EXMPCONF
+ Read . . . . . : 2 Read time . . . . . : 0.001792
+ Write . . . . . : 0 Write time . . . . . : 0.000000
+ Update . . . . . : 0 Update time . . . . . : 0.000000
+ Delete . . . . . : 0 Delete time . . . . . : 0.000000
+ Browse . . . . . : 0 Browse time . . . . . : 0.000000
+ Misc request . . . . . : 0 Misc request time . . . : 0.000000

```

Figure 10-53 File detail for a task

This level of detail shows, for each file accessed, the type of request and the response time to the application for those requests.

Controlling Transaction History

Transaction History is controlled through the global data area. The enablement of the feature is controlled in the <STARTUP_CONTROL> section, as shown in Example 10-3.

Example 10-3 Enabling the Transaction History feature

```

*
<STARTUP_CONTROL>
*
ONLINE_DATA_VIEWING=AUTO

```

The `ONLINE_VIEWER` section specifies the parameters for the subtask. Example 10-4 shows possible value configurations.

Example 10-4 Configuring values

```
*
<ONLINE_VIEWER>
DATA_STORE_TYPE=FILEOCMP
DATA_STORE_FILE_NAME=OMEGAXE.SC66.*.RKC2HIST
EXCLUDED_TRANS=(KD40,KD4C,KD4,KD4D,CSSY)
*

or

*
<ONLINE_VIEWER>
DATA_STORE_TYPE=DSPACE
DATA_STORE_SIZE=956
RESERVED_SIZE=25
EXCLUDED_TRANS=(KD40,KD4C,KD4,KD4D,CSSY)
```

Notes about Example 10-4:

- ▶ The `DATA_STORE_TYPE` keyword specifies how the data is to be stored. There are two options:
 - `FILEOCMP` is where the data is stored to a VSAM Linear data set. This option allows the data to persist if the CICS region or the KOCCL is shutdown.
 - `DSPACE` is where a data space is allocated to the KOCCL address space to hold the data. This option does not persist the values.
- ▶ The `DATA_STORE_FILE_NAME` is only applicable to a type of `FILEOCMP`. If the name provided contains an asterisk, as in Example 10-4, the asterisk is replaced with the JOB name of the CICS region.
- ▶ The `DATA_STORE_SIZE` and `RESERVED_SIZE` are only applicable to a type of `DSPACE`. The size refers to the size of the data space in kilobytes and the reserved size is the percentage of the space that is reserved for file details and application trace information.
- ▶ The `EXCLUDED_TRANS` keyword is applicable to both types and specified transactions that are not to be recorded to history.

Application trace

In addition to file and database detail statistics, OMEGAMON offers the capability to trace the application calls that a transaction makes. OMEGAMON traces calls made through the EXEC interface, the resource manager interface, and from third-party database providers, such as ADABAS, SUPRA, DATACOM, and IDMS.

The Application Trace display can be linked in OMEGAMON XE for CICS workspaces that show transaction History data. These are the Online Data Viewing, Units of Work, and the Web Services Transactions display. Figure 10-54 on page 282 shows the Application Trace workspace.

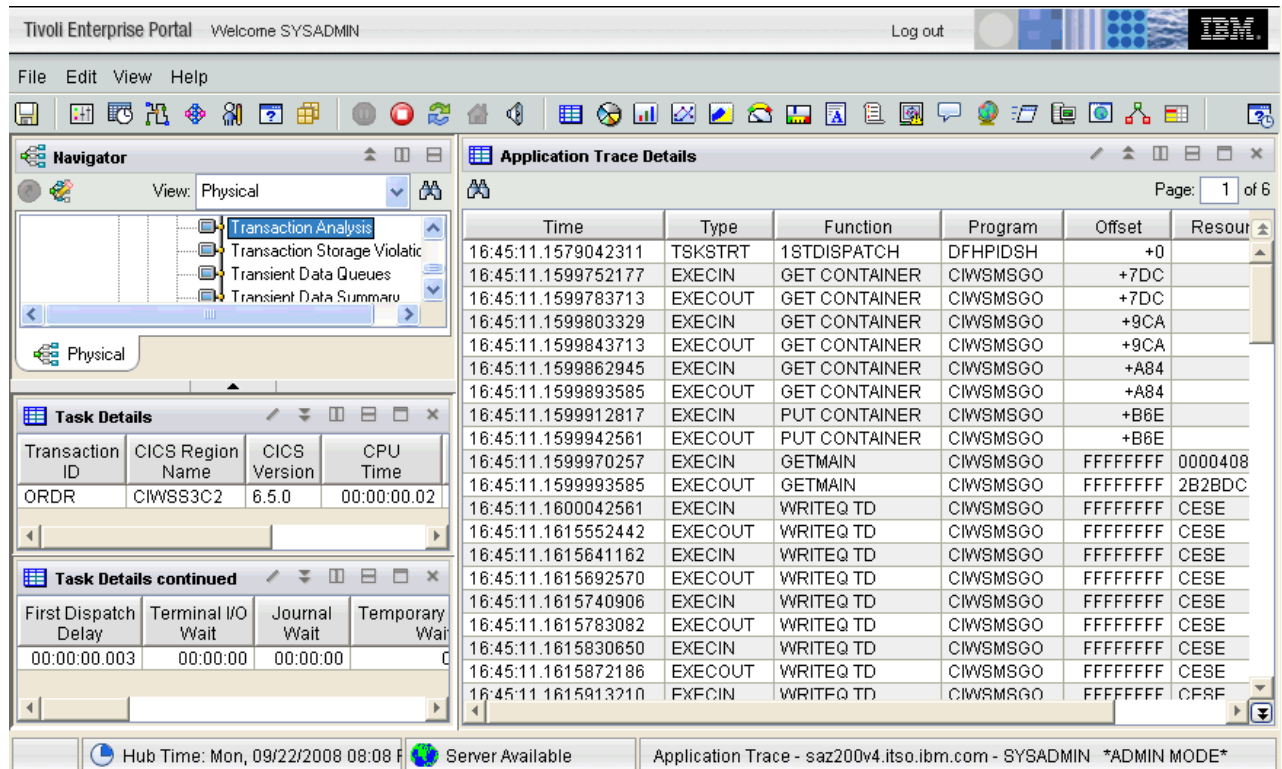


Figure 10-54 Application Trace workspace

The Application Trace workspace displays the trace data for a task to the right of the window. To the left, are details from the transaction history record.

Similar information is displayed in the menu system. You can display Figure 10-55 on page 283 by selecting the zoom key on the Application Trace Active heading in the transaction detail display. Zooming on the task from the Transaction History Trace record view also displays application trace (H.C).

ZZONDA VTM CIWSS3C2 V560./C SC66 09/22/08 20:12:29						
> PF1 Help PF3 Back PF4 Main Menu PF7 Up PF8 Down						
=====						
> TRACED TRANSACTION SUMMARY						
ONDV 05 TRACE SUMMARY						
+ Transaction Detail for ORDR task number =78877						
+						
+						
+ Application Trace Facility						
+ Lines 1 to 243 of 55						
+ Trace Type Program Offset Function Resource Response Elapsed						
+ Time						
+ -----						
+ TSKSTRT	DFHPIDSH	0	1ST DISPATCH			
+ EXECIN	CIWSMSG0	7DC	GET CONTAINER			0.00006
+ EXECOUT	CIWSMSG0	7DC	GET CONTAINER		NORMAL	0.00000
+ EXECIN	CIWSMSG0	9CA	GET CONTAINER			0.00000
+ EXECOUT	CIWSMSG0	9CA	GET CONTAINER		NORMAL	0.00000
+ EXECIN	CIWSMSG0	A84	GET CONTAINER			0.00000
+ EXECOUT	CIWSMSG0	A84	GET CONTAINER		NORMAL	0.00000
+ EXECIN	CIWSMSG0	B6E	PUT CONTAINER			0.00000
+ EXECOUT	CIWSMSG0	B6E	PUT CONTAINER		NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	GETMAIN	00004080		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	GETMAIN	2824DC58	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	FFFFFF	WRITEQ TD	CESE		0.00000
+ EXECOUT	CIWSMSG0	FFFFFF	WRITEQ TD	CESE	NORMAL	0.00000
+ EXECIN	CIWSMSG0	C52	GET CONTAINER			0.00000
+ EXECOUT	CIWSMSG0	C52	GET CONTAINER		NORMAL	0.00000

Figure 10-55 Application Trace Menu System display

Application trace is a high overhead feature of OMEGAMON CICS. We recommend that you turn it on dynamically when required. Application trace can be controlled through the Control section of the Menu System. Specifically, you can use menu option O.W to activate and deactivate trace. Use menu option O.V to specify trace filters, which allow for trace to be active only for certain transactions.

Enterprise Java Monitoring

OMEGAMON XE for CICS provides monitoring for CICS Enterprise Java as part of the resource monitoring that we described in “Resource monitoring” on page 266. The primary workspace for monitoring is the Enterprise Java Analysis branch under a CICS region. Figure 10-56 on page 284 shows an example of the Enterprise Java Analysis workspace.

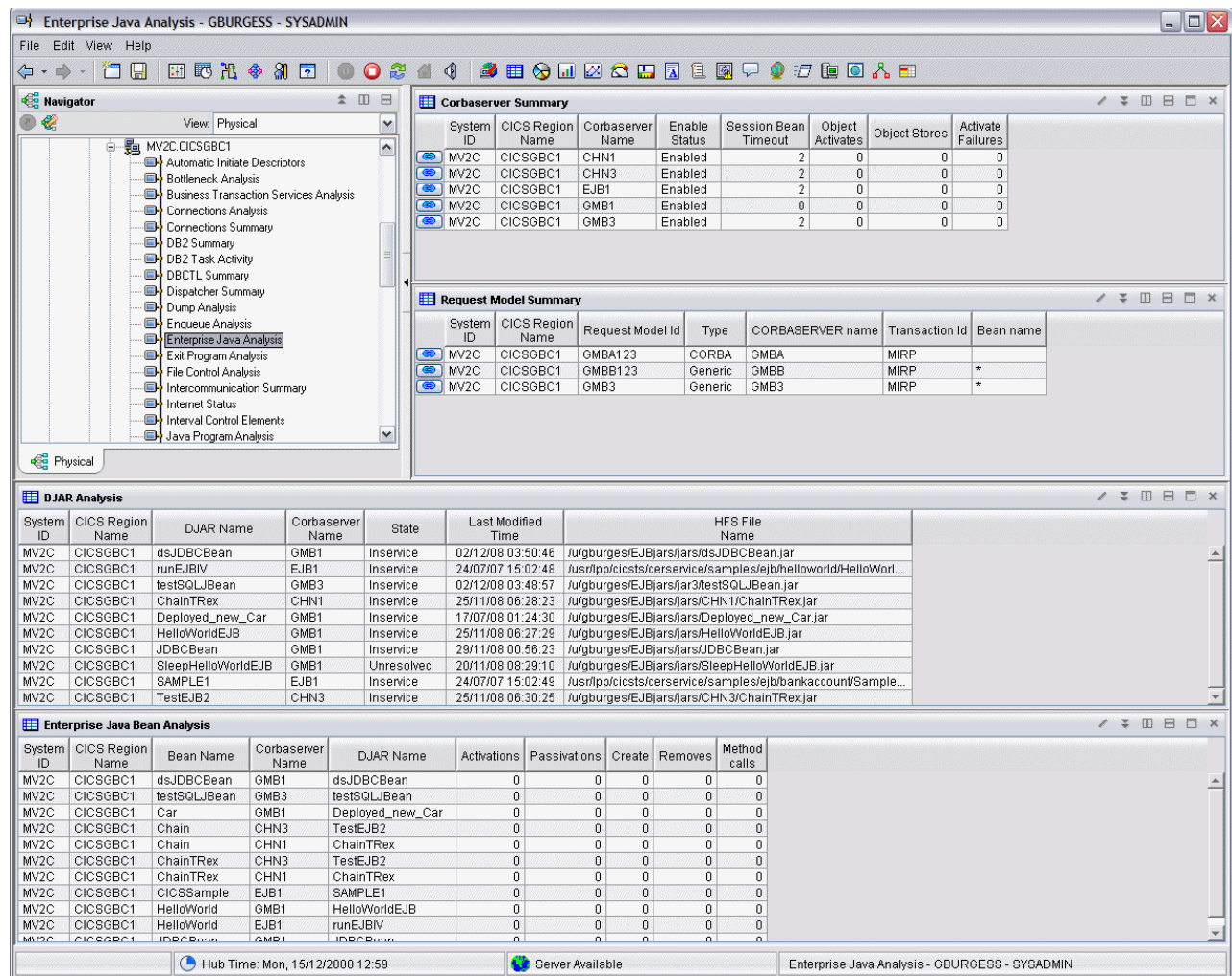


Figure 10-56 Enterprise Java Analysis workspace

From the Java Analysis workspace, you can see summary information that relates to the Enterprise Java resources in a particular CICS region. This workspace shows the following resources defined in CICS:

- ▶ Corbaserver Summary
- ▶ Request Model Summary
- ▶ DJAR Analysis
- ▶ Enterprise Java Bean Analysis

The Corbaserver and Request Model resources contain links to more details:

- ▶ For Corbaservers, the link displays more details about the resource, for example, the TCP/IP Service name and JNDI details.
- ▶ For Request Models, the link shows more details about the resource, such as Corbaserver, Transaction Identifier, Bean name, Interface, Module, and Operation.

Java Program Analysis

OMEGAMON XE for CICS provides monitoring for CICS Java Programs as part of resource monitoring. The primary workspace for monitoring is the Java Program Analysis branch under a CICS region. Figure 10-57 on page 285 is an example of the Java Program workspace.

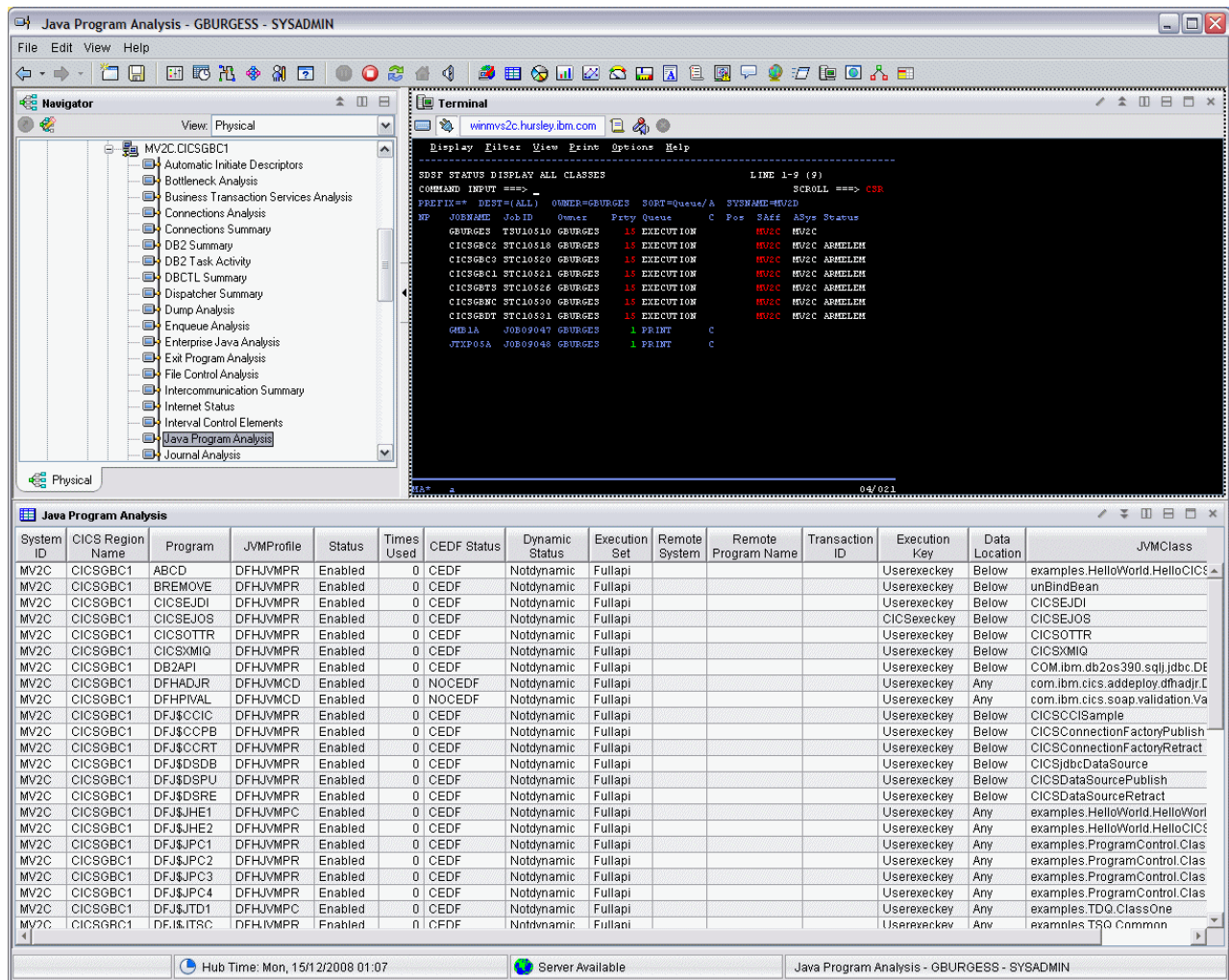


Figure 10-57 Java Program Analysis workspace

From the Java Program Analysis workspace, you can see information that relates to the Java Program resources in a particular CICS region.

Of particular interest is the JVM class that is associated with the program.

JVM Analysis

OMEGAMON XE for CICS provides monitoring for CICS JVMs. The primary workspace for monitoring is the JVM Analysis branch under a CICS region. Figure 10-58 on page 286 shows an example of the JVM Analysis workspace.

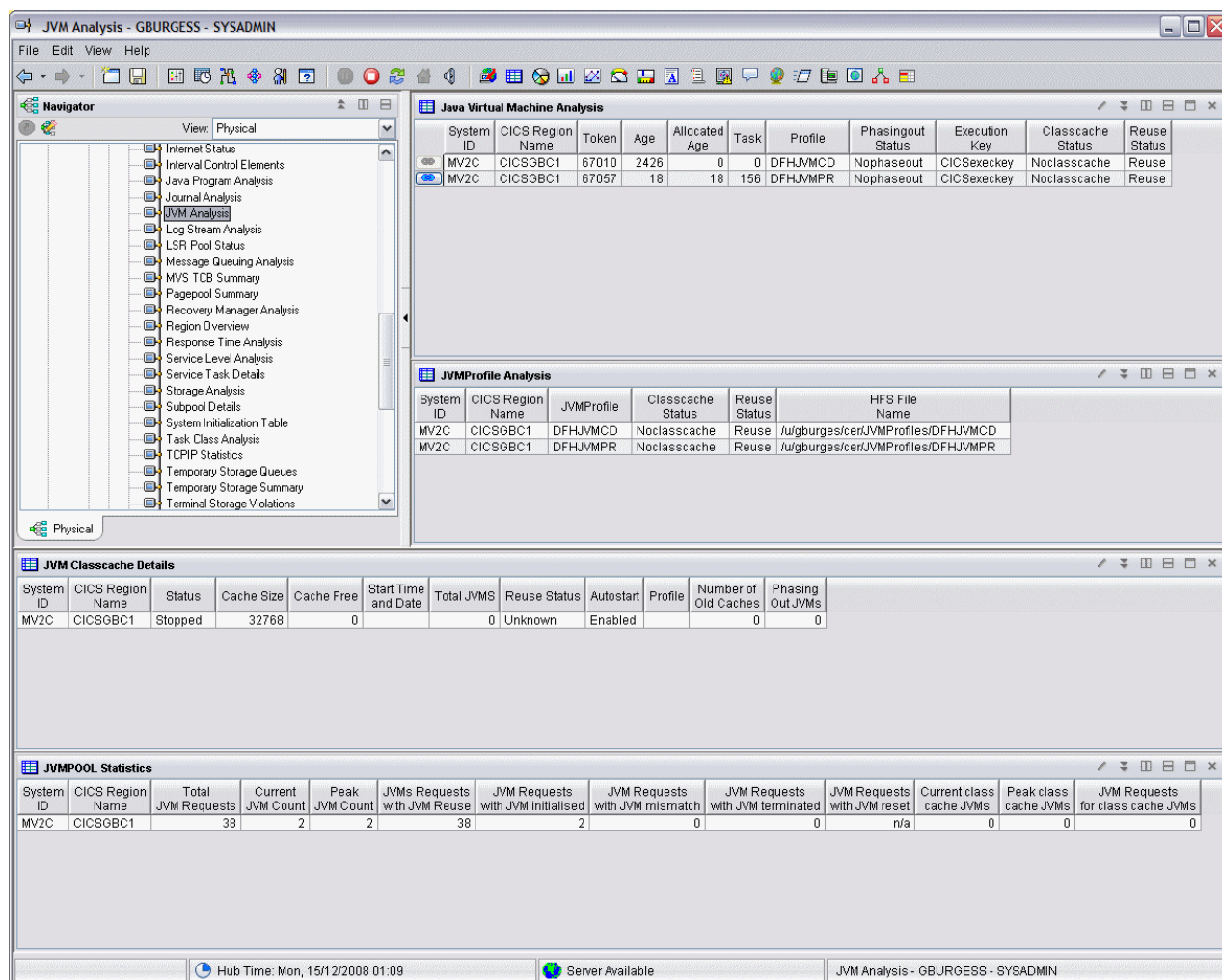


Figure 10-58 JVM Analysis workspace

From the JVM Analysis workspace, you can see information that relates to the JVM resources in a particular CICS region. This workspace shows the following resources:

- ▶ Java Virtual Machine Analysis
- ▶ JVM Profile Analysis
- ▶ JVM Classcache Details
- ▶ JVMPool Statistics

The Java Virtual Machine Analysis pane has a Dynamic Workspace Link to the Single Transaction Analysis Workspace, shown in Figure 10-59 on page 287. The link is greyed out when a JVM is not associated with a transaction.

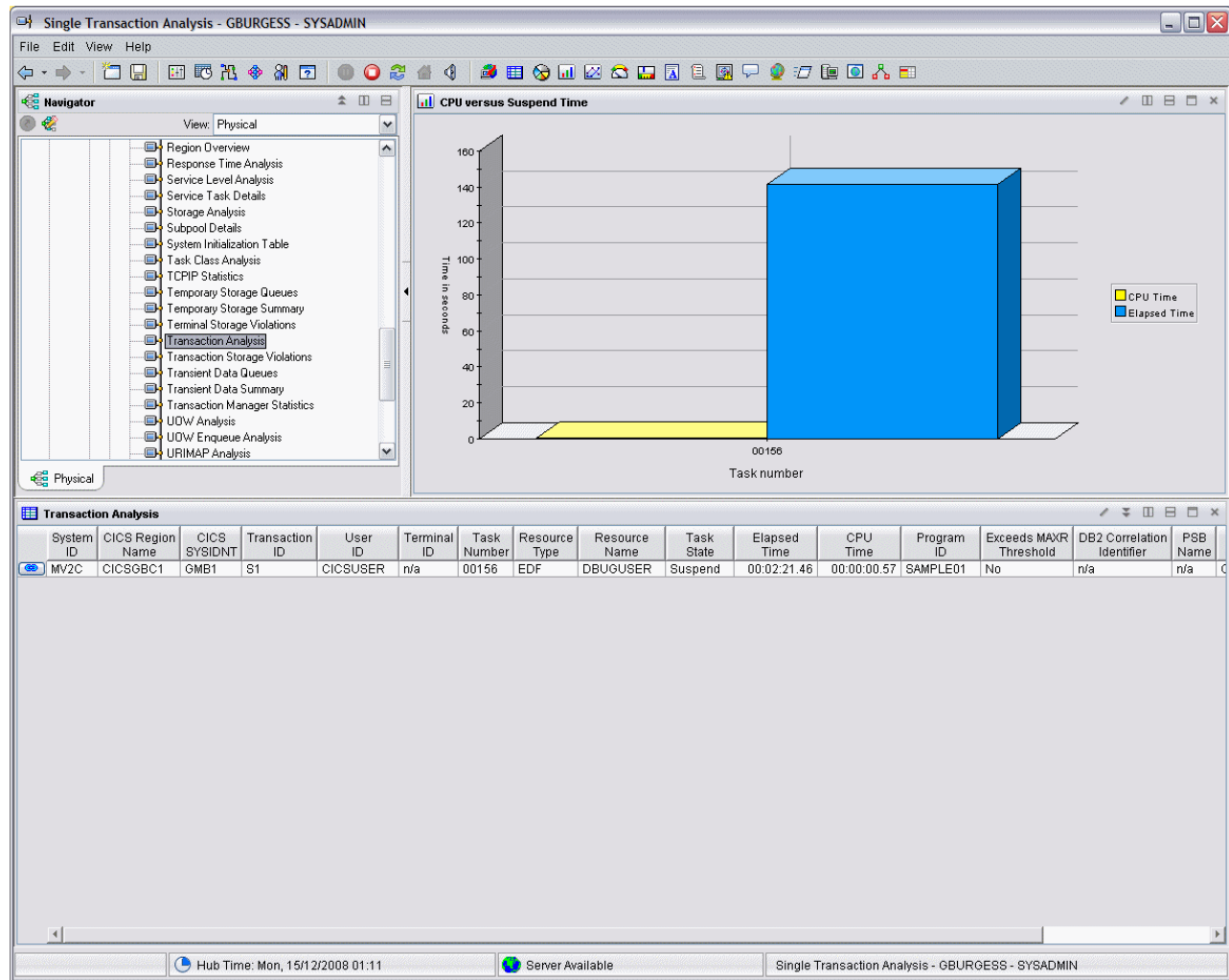


Figure 10-59 Single Transaction Analysis Workspace

This workspace is identical to the Transaction Analysis workspace. The only difference is that it is filtered using the task number of the transaction that is executing the JVM. All the Links that are associated with a transaction in the Transaction Analysis workspace are available to the single transaction instance.

Resource limiting

Resource limiting (RLIM) is a special feature that OMEGAMON CICS provides. This feature is designed to automatically protect the environment from rogue CICS transactions that might loop in a way that cannot be detected by the CICS runaway task protection or transactions that use extraordinary amounts of certain resources.

Resource limiting examines the values for certain resources or the number of certain types of requests. If the specified thresholds were exceeded, OMEGAMON either issues a message or causes the transaction to abnormally terminate (ABEND).

The enablement of the feature is controlled in the <STARTUP_CONTROL> section, as shown in Example 10-5.

Example 10-5 Enabling Resource limiting

```
*
  <STARTUP_CONTROL>
*
  RESOURCE_LIMITING=AUTO
  RESOURCE_LIMITING_MSG_DEST=TDQ
  RESOURCE_LIMITING_SYSTEM_TASKS=NO
  RESOURCE_LIMITING_ABEND_CANCEL=YES
```

Notes about the options in Example 10-5:

- ▶ The option RESOURCE_LIMITING_MSG_DEST specifies if the message is to be written to the transient data queue CSSL (TDQ) or to the system console (LOG).
- ▶ RESOURCE_LIMITING_SYSTEM_TASKS allows you to specify if resource limiting is to issue warning messages for CICS system transactions. OMEGAMON will not cause a system task to ABEND.
- ▶ RESOURCE_LIMITING_ABEND_CANCEL allows you to specify whether any application abend handling exits are to be honored when RLIM elects to ABEND a transaction. Specifying NO indicates that ABEND exits will not be cancelled and therefore will remain in effect.
- ▶ The <RESOURCE_LIMITING> section of the global specifies the limits that are to be in effect for a given transaction ID, as shown in Example 10-6.

Example 10-6 <RESOURCE_LIMITING>

```
*
  <RESOURCE_LIMITING>
*
  <<CPU>>
  INCLUDED_TRANS=(TRLN,DE*)
  KILL_LIMIT=10
  WARN_LIMIT=5
*
  <<VSAM>>
  INCLUDED_TRANS=(STRS,F?L?,TRLN)
  KILL_LIMIT=100
  WARN_LIMIT=50
*
```

In Example 10-6, transactions TRLN has the message OC8902 issued, if it makes more than 50 EXEC CICS requests against a FILE or if the CPU used exceeds five seconds. If the transaction exceeds 10 seconds of CPU or 100, EXEC CICS FILE requests the message OC8903 get issued and the transaction is abended.

Specifying a question mark (?) in the transaction ID indicates that the rule matches for a transaction ID that has any character in that location. So a transaction ID of FALZ has WARN and KILL limits of 50 and 100 for VSAM. Transaction ID FLAZ will not.

An asterisk (*) can be specified at the end of a transaction ID to indicate any trailing characters will match.



Part 4

Appendix



JCICS exception mapping

Table A-1 shows the mapping of the exceptions that JCICS commands can throw to the CICS conditions that they represent. These mappings can be particularly useful when you are trying to understand the meaning of a JCICS exception.

Table A-1 Mapping

CICS Condition	Java exception	CICS condition	Java exception
ALLOCERR	AllocationErrorException	CBIDERR	InvalidControlBlockId
CCSIDERR	CCSIDErrorException	CHANNELERR	ChannelErrorException
CONTAINERERR	ContainerErrorException	DISABLED	FileDisabledException
DSIDERR	FileNotFoundException	DSSTAT	DestinationStatusChangeException
DUPKEY	DuplicateKeyException	DUPREC	DuplicateRecordException
END	EndException	ENDDATA	EndOfDataException
ENDFILE	EndOfFileException	ENDINPT	EndOfInputIndicatorException
ENQBUSY	ResourceUnavailableException	ENVDEFERR	InvalidRetrieveOptionException
EOC	EndOfChainIndicatorException	EODS	EndOfDataSetIndicatorException
EOF	EndOfFileIndicatorException	ERROR	ErrorException
EXPIRED	TimeExpiredException	FILENOTFOUND	FileNotFoundException
FUNCERR	FunctionErrorException	IGREQID	InvalidREQIDPrefixException
IGREQCD	InvalidDirectionException	ILLOGIC	LogicException
INBFMH	InboundFMHException	INVERRTERM	InvalidErrorTerminalException
INVEXITREQ	InvalidExitRequestException	INVLDC	InvalidLDCEException
INVMPsz	InvalidMapSizeException	INVPARTNSET	InvalidPartitionSetException
INVPARTN	InvalidPartitionException	INVREQ	InvalidRequestException
INVTsREQ	InvalidTSRequestException	IOERR	IOException

CICS Condition	Java exception	CICS condition	Java exception
ISCINVREQ	ISCInvalidRequestException	ITEMERR	ItemErrorException
JIDERR	InvalidJournalIdException	LENGERR	LengthErrorException
MAPERROR	MapErrorException	MAPFAIL	MapFailureException
NAMEERROR	NameErrorException	NODEIDERR	InvalidNodeIdException
NOJBUFSP	NoJournalBufferSpaceException	NONVAL	NotValidException
NOPASSBKRD	NoPassbookReadException	NOPASSBKWR	NoPassbookWriteException
NOSPACE	NoSpaceException	NOSPOOL	NoSpoolException
NOSTART	StartFailedException	NOSTG	NoStorageException
NOTALLOC	NotAllocatedException	NOTAUTH	NotAuthorisedException
NOTFND	RecordNotFoundException	NOTOPEN	NotOpenException
OPENERR	DumpOpenErrorException	OVERFLOW	MapPageOverflowException
PARTNFAIL	PartitionFailureException	PGMIDERR	InvalidProgramIdException
QBUSY	QueueBusyException	QIDERR	InvalidQueueIdException
QZERO	QueueZeroException	RDATT	ReadAttentionException
RETPAGE	ReturnedPageException	ROLLEDBACK	RolledBackException
RTEFAIL	RouteFailedException	RTESOME	RoutePartiallyFailedException
SELNERR	DestinationSelectionErrorException	SESSBUSY	SessionBusyException
SESSIONERR	SessionErrorException	SIGNAL	InboundSignalException
SPOLBUSY	SpoolBusyException	SPOLEERR	SpoolErrorException
STRELERR	STRELERRException	SUPPRESSED	SuppressedException
SYMBOLERR	SymbolErrorException	SYSBUSY	SystemBusyException
SYSIDERR	InvalidSystemIdException	TASKIDERR	InvalidTaskIdException
TCIDERR	TCIDERRException	TEMPLATERR	TemplateErrorException
TERMERR	TerminalException	TERMIDERR	InvalidTerminalIdException
TOKENERR	TokenErrorException	-----	-----
TRANSIDERR	InvalidTransactionIdException	TSIOERR	TSIOErrorException
UNEXPIN	UnexpectedInformationException	USERIDERR	InvalidUserIdExceptio
WRBRK	WriteBreakException	WRONGSTAT	WrongStatusException



B

Hints and tips

In this Appendix, we provide some tips through the evolution of what a typical Java program in CICS might look like.

Priority of public static void main() methods

CICS allows you to use two alternative main() methods for executing Java programs:

- ▶ public static void main(CommAreaHolder commAreaHolder)
- ▶ public static void main(String[] args)

The first passes COMMAREA data in a CommAreaHolder, which holds the data in a byte array. If this version of the main() method does not exist, CICS attempts to call the “typical” second version (note that with this main() method, no data is passed with the arguments and so the supplied String array is empty).

Getting transaction arguments using Java

When a Java program in CICS is executed by typing a transaction definition in the terminal, it is possible to pass values after the transaction name for use as parameters in the program, as shown in Example B-1. Using the JCICS implementation of EXEC CICS RECEIVE, the terminal data is got as a byte array and wrapped in a String object before returning.

Example: B-1 Return transaction arguments as a String

```
public static String getTerminalString() {
    final DataHolder dataHolder = new DataHolder();
    final Object principalFacility = Task.getTask().getPrincipalFacility();

    String result = null;

    if (principalFacility instanceof TerminalPrincipalFacility) {
        try { ((TerminalPrincipalFacility)principalFacility).receive(dataHolder); }
        catch (EndOfChainIndicatorException e) {}
        catch (Exception e) { System.err.println(e.toString()); e.printStackTrace(); }
    }

    return dataHolder.value != null ? new String(dataHolder.value) : null;
}
```

Using this facility, you can go a step further and *chop up* the parameters and return them in a String array. The default separating character between the parameters is the “ ” (space) character. Because the transaction name is also part of the String data, this becomes the first element in the array, as shown in Example B-2.

Example: B-2 Separate transaction arguments into a String array

```
public static String[] getTerminalStringAsArray() {
    String[] result = null;
    int count = 0;

    final String args = getTerminalString();

    if (args != null) {
        final java.util.StringTokenizer tokenizer =
            new java.util.StringTokenizer(args, " "); // (1)

        result = new String[tokenizer.countTokens()];

        while (tokenizer.hasMoreTokens()) result[count++] = tokenizer.nextToken();
    }
}
```



```
    return result;  
}
```

Notes on Example B-2 on page 294:

- To include spaces in the resulting string array elements, change ‘ ‘ to something like ‘,’.

Never use `System.exit()`

When Java applications are run in CICS, the `public static void main()` method is called through the use of another Java program called the Java wrapper (that is how functionality like that described in “Priority of `public static void main()` methods” on page 294 can happen). The use of the wrapper allows CICS to initialize the environment for Java applications, but more importantly, it performs cleanup for any processes that are used during the life of the application. Killing the JVM, even with a “clean” return code of 0, does not allow this cleanup process to run, and can lead to data inconsistency.

From an additional perspective, by using a `System.exit()` call, the continuous and resettable JVM modes become unusable because it terminates the JVM instance.

The recommended and only approach is to allow the program to run to the end of the `public static void main()` method and the JVM to terminate cleanly.



Resettable JVM

The resettable JVM was introduced in CICS TS Version 2.1. With it you could reset the state of a JVM between tasks to help ensure that subsequent users of the same JVM are fully isolated from states left behind by previous users of the JVM.

The time taken to reset a JVM depended on no cross-heap references between the middleware and application heaps. If references exist, the JVM scan to determine if the references are in live objects can be slow. If the attempt to reset the JVM failed, CICS discarded the JVM and created a new one. These unresettable events (UREs) were a major performance problem for some users of CICS TS and Java.

Support for the resettable JVM was removed in CICS TS 3.2. Anyone using a resettable JVM in earlier versions of CICS are encouraged to migrate to a continuous mode JVM.

The information in this appendix is for historical reference only.

Resettable JVM

The resettable JVM can be started and used to run consecutive Java applications, resetting its storage areas between each program invocation. This mode of operation offers the same isolation as the single use mode, which means that it prevents changes to the state of the JVM made by an application program affecting the operation of the next program run in that same JVM. However, it is much more efficient than single use mode, for these reasons:

- ▶ It avoids the overhead of destroying and starting the JVM.
- ▶ Classes and JIT-compiled code can be shared between JVMs, thus reducing I/O (for loading classes) and CPU time (for JIT compilation).

The JVM divides classes into categories of *well-behavedness* or *trustworthiness*, and it also monitors the behavior of application classes. If an application class changes the JVM state in a way that could potentially affect the next program run on that JVM, the JVM is marked unresettable and is destroyed after the program terminates.

The storage heap is divided up into a number of areas, each with its own class loader. These areas are treated differently during reset processing.

There is a transient heap, which contains objects that a Java application creates as it executes. All of these objects are deleted at reset time. There is also a separate trusted middleware heap that is used for middleware classes that are trusted to reset the object they create to some known state, as part of reset processing.

Resetting the JVM involves a check to see if any pointers exist from the middleware heap to the application heap. If there are none, the application heap is thrown away, which is a very cheap garbage collection operation.

If pointers do exist, a search similar to full garbage collection occurs to determine whether the pointers are live or not.

Resets can also fail for a number of application-related reasons, such as the changing of a system property or the loading of a native library. If the reset operation fails to complete, CICS is informed of this and terminates the JVM. See “Unresettable actions” on page 298 for details.

In summary, the resettable JVM provides the benefits of reuse while offering the greatest protection between program invocations.

Unresettable actions

A resettable JVM must be discarded after an application program changes its state in a way that might affect another program that is running on that JVM. Therefore, to benefit from using the JVM in resettable mode, you must ensure that your applications stick to certain rules.

If an application program performs one of these actions, the JVM becomes unresettable:

- ▶ Writing to a static variable that is associated with a class that is loaded by either the bootstrap class loader or the extensions class loader
- ▶ Setting system properties:
 - `java.lang.System.setProperty()`
 - `java.lang.System.setProperties()`
 - `java.lang.System.getProperties()`

The first two are special cases of the modification of a static variable (in effect, they are write accesses to the static variable `System.props`).

- ▶ Closing or redirecting the standard input, output, or error streams:
 - `System.err.close()`, `System.in.close()`, `System.out.close()`
 - `java.lang.System.setErr()`, `java.lang.System.setIn()`, `java.lang.System.setOut()`

Again, these are special cases of writing to a global static variable.

- ▶ Loading a native library:
 - `java.lang.System.loadLibrary()`
 - `java.lang.System.load()`
 - `java.lang.Runtime.loadLibrary()`
 - `java.lang.Runtime.load()`

Because the JVM cannot know if native library code changes its state in one of the other ways described here, it marks itself as being unresettable.

- ▶ Using the Abstract Windowing Toolkit (AWT) or any of its derivatives to access a display or keyboard, or to print.
- ▶ Several security-related actions, such as setting the security manager or attempting to access or modify the security configuration objects (Policy, Security, Provider, Signer, Identity).
- ▶ Using any of the thread methods to start, stop, resume, suspend, or interrupt threads.
- ▶ Creating a new process using `Runtime.exec()`.
- ▶ Loading a non-checked standard extension.

Checked standard extensions are those extensions that were checked for writable statics. The writable statics are either not exposed to application code or checks were added to the standard extension to detect any changes that are made.

Checked extensions are identified by the presence of a manifest entry in the main section of their JAR files:

```
IBM-Reusable-JVM-Compatible: True
```

- ▶ Using the JVM in debug mode (Xdebug option).

Checking for reset behavior

CICS provides logging facilities to assist you in obtaining whether and why an application program caused a JVM to become unresettable.

Tip: This facility is not only useful for making your applications resettable-friendly, it is also a good start for migrating your applications from the resettable JVM to the continuous JVM. Many of the events that render a JVM unresettable can be potential problems when running in continuous mode.

Logging the events that cause a JVM to be marked as unresettable is controlled through the system properties file:

```
ibm.jvm.events.output=log file name
```

```
ibm.jvm.unresettable.events.level=max (Use this setting for development/test.)
```

or

```
ibm.jvm.unresettable.events.level=min (Use this setting for production.)
```

The three events to look out for are:

- ▶ **UnresettableEvent**

This is the worst type because it means that the JVM is marked unresettable and must be terminated. A textual description of the event tells you why that happened, and you will see a stack trace that tells you where in the application code it happened.

- ▶ **ResetTraceEvent**

This event means that there are cross-heap references from middleware objects to application objects, which is still bad, but not as bad as an `UnresettableEvent` because it does not necessarily mean that the JVM must be terminated.

For an in-depth discussion about `ResetTraceEvents` and how to get rid of them, refer to *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201.

- ▶ **ResetJVMEvent**

This is the good one because it means that the JVM reset cleanly.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 302. Note that some of the documents referenced here are available in softcopy only.

- ▶ *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177
- ▶ *ABCs of z/OS System Programming Volume 9*, SG24-6989
- ▶ CICS Transaction
- ▶ *Server V3R1 Channels and Containers Revealed*, SG24-7227-00

Other publications

These publications are also relevant as further information sources:

- ▶ *Java Applications in CICS*, SC34-6825
- ▶ *z/OS V1R9.0 UNIX System Services Planning*, GA22-7800
- ▶ *z/OS MVS Initialization and Tuning Reference*, SA22-7592
- ▶ *CICS Performance Guide*, SC34-6833
- ▶ *CICS System Definition Guide*, SC34-6813
- ▶ *IBM Developer Kit And Runtime Environment, Version 5.0 Diagnostics Guide*, SC34-6650
- ▶ *CICS Resource Definition Guide*, SC34-6815
- ▶ *CICS Supplied Transactions*, SC34-6817
- ▶ *CICS System Programming Reference*, SC34-6820
- ▶ *WUI in CICSplex SM Web User Interface Guide*, SC34-6841
- ▶ *CICS Messages and Codes*, GC34-6241
- ▶ *CICS Intercommunication Guide*, SC34-6243
- ▶ *JZOS Installation and User's Guide*, SA23-2245
- ▶ *CICS Application Programming Guide*, SC34-6231
- ▶ *CICS Internet Guide*, SC34-6245
- ▶ *CICS Problem Determination Guide*, SC34-6239
- ▶ *CICS Messages And Codes*, GC34-6442

Online resources

These Web sites are also relevant as further information sources:

- ▶ Java SDKs for z/OS
<http://www-03.ibm.com/servers/eserver/zseries/software/java/allproducts.html>
- ▶ IBM CICS SupportPac site:
<http://www-1.ibm.com/support/docview.wss?rs=1083&uid=swg27007241>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Numerics

3270 151
64-bit storage 20
64-bit Support 5

A

AddressResource 113
Analysis point monitoring (APM) 246
APF authorized 261
ASCII & EBCDIC issues 87
ASKTIME 165
Assembler 83
Autostartst 30

B

Basic Mapping Support 93
Binary based values 141
BMS 93, 151
BPXI040 45
BPXPRMxx 45
BPXPRMxx parameter 45
 MAXASSIZE 45
 MAXCPUIME 45
 MAXFILEPROC 45
 MAXPROCUSER 45
 MAXTHREADS 45
BPXPRMxx. 45
BUY-SELL-UPDATE 169

C

C/C++ 4
Cachefree 30
Cachesize 30
CALCULATE-SHARES-BOUGHT 168
calculateSharesBought() 168
CECI. 52
CEDA 44, 52
CEEDUMP 202
CEEHRNUH 201
CEEPIPI 44, 195
CEMT INQUIRE CLASSCACHE 53
CEMT INQUIRE DISPATCHER 54
CEMT INQUIRE JVM 54
CEMT INQUIRE JVMPPOOL 55
CEMT INQUIRE PROGRAM 55
CEMT PERFORM CLASSCACHE 56
CEMT SET DISPATCHER 58
CEMT SET JVMPPOOL 58
CEOT 52
Channels and Containers 20
Character based values 141
CHAR-VALUE 169

CICS Business Transaction Services (BTS) 238
CICS directory 24
CICS Dynamic Storage Areas (DSAs) 47
CICS Explorer 233–234
CICS JVM plug-in mechanism 218
CICS language interface module 90
CICS LINK 101
CICS Monitoring Facility (CMF) 238
CICS PA 237
CICS PROGRAM definition attributes 52
 CONCURRENCY 53
 JVM 52
 JVMCLASS 52
 JVMPROFILE 52
 LANGUAGE 53
CICS terminal 44
CICS Transaction Gateway server 5
CICS translator 90
CICS using HTTP 114
CICS VSAM Transparency 173
CICS Web Support 238
CICS Web support 114
CICSplex System Management 233
CICSUSER 156
Class declaration and constructors 117
class sharing 225
COBOL 6, 136
Collecting statistics using CICSplex SM monitoring 246
COMMAREA 103–104, 139, 154
COMMAREA-BUFFER 140, 163
CommareaWrapper 140, 155, 163
Common Work Area 94
Common Work Area (CWA) 94
COMPANY-IO-BUFFER 163
CompanyIOWrapper 163
COMPFILE 136
Component 126
Container 127
Continuous JVM 222
conversion 163
Corbaservers 249
CUSTFILE 136
CustomerIOWrapper 163
CWBA 148

D

Data migration SQL 173
Database tables 172, 179–180, 182, 189
Datestarted 31
DB2 5, 161
Debugging 194
 JVM abended? 195
 JVM failing to start? 195
Debugging the application 216

- DecimalFormat 169
- Deploying code to CICS 69
- DFH\$JVM 44
- DFHISOUtil 24
- dfhjaiu.jar 24
- DFHJVMCC 51
- DFHJVMCD 48, 51
- DFHJVMP 51
- DFHJVMPR 48, 51
- DFHJVMP 51
- DFHJVMRO 46
- DFHLS2WS 179
- DFHWEB 150
- DFHWS2LS 186
- DFJ\$JHE2 44
- DOCTEMPLATE 156–157
- Document 149
- Document API 92
- Document services 92, 114
- Documents 153
- dynamic HTML 153
- Dynamic VIPA (virtual IP addressing) 6

E

- EIBRESP 164
- EJBs 6
- ENQ/DEQ 112
- ENQMODEL 114
- Enterprise JavaBeans 4
- Entry Sequenced Data Sets (ESDS) 92
- entry-sequenced data set 115
- Entry-sequenced data set (ESDS) 115
- ESDS 115
- EVALUATE 166
- Example 6-32 179, 181, 185, 187–188
- exceptions
 - checked 97
 - unchecked 97
- Excessive garbage collection 225
- EXEC CICS 83
- EXEC CICS DOCUMENT 114
- EXEC CICS WEB 114

F

- File I/O 86
- Files 136
- formatting 163
- Formatting trace 209
- FTP 66

G

- garbage collection 227
- GET 153
- getByteArray() 149
- GET-COMPANY 166–167
- getCompany() 166, 168
- GID 46
- GUI 18

H

- Heap management 34
- HelloCICSWorld Java application 44
- HFS 42
- High Performance Java 18
- HiperSockets 4
- Hot Pooling 18
- HPJ 18
- HTML 4, 151
- HTML design 152
- HTTP input fields 153
- HttpResponse 149

I

- IBM Directory Server 5
- IBM eServer zSeries 4
- ImageLoader 158
- IMS(TM) 6
- Integrated Cryptographic Service Facility (ICSF) 5
- Intelligent Resource Director (IRD) 4
- IPCSEMNSEMS 44
- ISPF interface 38
- ISPF shell 39

J

- J8 CPU 244
- JAR files 248
- Java 162
- Java 1.4.2 shared class cache 28
- Java 2 Enterprise Edition (J2EE) 7
- Java 5 28
- Java 5 shared class cache 28
- Java Build Path 67
- Java Language 7
- Java program 25
- Java Program Objects 19
- Java servlets 93
- Java Software Development Kit (SDK), available products
 - on z/OS 10
- Java Virtual Machine 9
- JAVA_DUMP_OPTS 200
- Javadumps 199
- JBMS 93
- JCICS 19
- JCICS API 91
- JCICS Web interface 138, 151
- JCICS Web program 148
- JHE2 44
- JNDI 21
- Just In Time Compiler (JIT) 7
- JVM 18
- JVM Class Caches 236
- JVM class loader tracing 209
- JVM method tracing 207
- JVM operation modes 20
 - Continuous mode 20
 - Resettable mode 20
 - Single use mode 20
- JVM Pool 236

- JVM profile 37, 43
- JVM profile options 50
 - CICS_DIRECTORY 50
 - CLASSCACHE 50
 - CLASSPATH 50
 - JAVA_HOME 50
 - JVMPROPS 50
 - LIBPATH 50
 - REUSE 50
 - STDERR 50
 - STDIN 50
 - STDOUT 50
 - WORK_DIR 50
 - Xinitacsh 50
 - Xinitsh 50
 - Xinitth 50
 - Xmaxe 50
- JVM profile options table 50
- JVM Profiles 237
- JVM properties file 37
- JVM Status 237
- JVM stealing 225
- JVM support in CICS 18
 - CICS TS 1.3 18
 - CICS TS 2.1 and 2.2 19
 - CICS TS 2.3 19
 - CICS TS 3.1 19
- JVM system properties files 51
- JVMCCPROFILE 28, 48
- JVMCCSIZE 48
- JVMCCSTART 48
- JVMCCSTART PARAMETERS
 - JVMCCSTART=NO 29
 - JVMCCSTART=YES 29
- JVMPPOOL 55
- JVMPROFILEDIR 43, 47
- JVMSet 28
- JVMSets and the shared class cache
 - Benefits of the shared class cache 28
 - JVMCCSTART=AUTO,JVMCCSTARTPARAMETERS
 - JVMCCSTART=AUTO 29
 - JVMCCSTART=NO 29
 - JVMCCSTART=YES 29
 - Master JVM 28
 - Worker JVM 29
- JZOS 134
- JZOS APIs 134
- JZOS package 142
- JZOS_HOME 142

K

- Key Sequenced Data Sets (KSDS) 92
- KEY8 CPU 244
- key-sequenced data set 115
- Key-sequenced data set (KSDS) 115
- KSDS 115

L

- Language Environment 45
 - CEEDEOPT 46
 - DFHJVMRO 46
 - SCEERUN 45
 - SCEERUN2 45
- Language Environment (LE) 44
- legacy trader application 136
- LIBPATH 47
- LINK 103
- Linux for zSeries 4
- logoff 161
- LPARs 5

M

- main() 165
- MAINLINE 165
- Mapset 137
- MAS resource monitoring (MRM) 246
- master JVM 28
- MAXASSIZE 44
- MAXCPU TIME 45
- MAXFILEPROC 44
- MAXJVMTCB 226
- MAXJVMTCBS 49
- MAXPROCSYS 44
- MAXPROCUSER 44–45
- MAXPTYs 44
- MAXTHREADS 45
- MAXTHREADTASKS 45
- MAXUIDS 44
- move 167
- MRO 238

N

- NameResource 113
- NEWTRAD 137
- non-shareable 29
- NUM-VALUE 169

O

- Object Request Broker (ORB) 87
- object-oriented (OO) 3
- OCCURS X TIME 141
- Oldcaches 31
- OMEGAMON XE for CICS on z/OS 233
- OMVS 43
- OpenLDAP 5

P

- Parallel Sysplex 4
- PERFORM 166–167
- performBuy() 154
- performSell() 154
- Phasingout 31
- PIPELINE 179
- PL/I 6

- POST 153
- PROCLIB 261
- Program control 91
- Programs 136
- PTF
 - PK59577 19
- PTFs 195
- Public Key Services in z/OS 6

R

- RACF ADDUSER 45
- RACF OMVS 45
 - ASSIZEMAX 45
 - CPUTIMEMAX 45
 - FILEPROCMAx 45
 - PROCUSERMAX 45
 - THREADSMAX 45
- RANDUMP 246
- Rational Application Developer 63
- Rational Developer for System z 63
- RBHOME 158
- real-time analysis (RTA) 246
- Redbooks Web site 302
 - Contact us xiii
- relative byte address (RBA) 115
- relative record data set 115
- Relative record data set (RRDS) 115
- Relative Record Data Sets (RRDS) 92
- request formats 148
- RETURN 167
- Reusable JVM 222
- Reusest 31
- rlogin 40
- RRDS 115

S

- SCEERUN 45
- SCEERUN2 45
- SDFHSAMP 46
- SDFJAUTH 47
- SDK tools 12
- sendErrorResponse() 150, 157
- sendLinkRequest() 156
- sendResponse() 156
- services oriented architecture (SOA) 6
- SETOMVS 45
- Shared Class Cache 225
- Shared Class Cache Facility 223
- showList() 155
- SimpleTraderPL 149
- SMF 238
- SMF 110 238
- Sockets 86
- SQL 173
- SQL statements 175
- ssh 40
- strip() 155
- symbol list 155
- SymbolList 157

- symbols 153
- SYSDDUMP 203, 246
- Sysplex Distributor 6
- System Availability Monitoring (SAM) 246
- System Display and Search Facility (SDSF) 134
- System Logger 238

T

- tail 166
- TCP/IP 70
- TCP/IP network support 6
- TCPIPS 150
- TCPIPService 150, 179
- telnet 40
- templates 153, 158
- Terminal control 93
- Terminal services 125
- Threads 85
- Timestarted 31
- TN3270 5
- Totaljvms 31
- TRADER 137
- Trader-1.jar 150
- TRADERBL 136
- TraderConstants 148, 154
- TraderDataAccess 164
- TRADERPJ 157, 191
- TRADERPL 136
- TraderPL 153
- TRADERPS 150
- Transaction Work Area 94
- Transaction Work Area (TWA) 94
- Transient data queues 93
- Transient storage queues 93
- Troubleshooting 74
- TSO ISHELL 39
- TSO OMVS 39

U

- UID 46
- Unicode 4
- Unix System Services (USS) 24
- Unknown RefID_ijava
 - bytecode 9
 - dynamically loading classes 9
- UPDATE-BUY-SELL 154
- URL 150, 158, 191
- userid 156
- USEROUTPUTCLASS 47
- Using Application Classpath 225

V

- VisualAge for Java 104
- VSAM 6
- VTAM 94

W

- Web 148

- Web and TCP/IP services 93
- Web pages 152
- Web security 160
- WEBSERVICE 179
- WebSphere MQ 138, 238
- WebSphere MQ message queues 93
- worker JVM 28
- Working storage 162
- WORKING-STORAGE 163
- wrapper 140
- writeMessage() 165
- WRITEQ-TS 165
- Writeq-TS 165
- Writing trace to buffers 208
- WSbind 180
- WSDL 180

X

- XCTL 103
- XCTL (“transfer control”) 101
- XML 4

Z

- z/Architecture 4
- z/OS 3, 235
 - 4
- z/OS HTTP Server 5
- z/OS LPAR 40
- z/OS Managed System Infrastructure 5
- z/OS UNIX shell 37
- z/OS UNIX System Services for z/OS (USS) 9
- z/VM 4
- zAAP
 - benefits 228
 - introduction 227
 - specialty engines 227
 - workflow 229
- zFS 41
- zFS directory 42
- zSeries Application Assist Processor 10



Java Application Development for CICS

(0.5" spine)
0.475" <-> 0.873"
250 <-> 459 pages



Java Application Development for CICS

Migrating to Java 5 and CICS TS 3.2

Determining CICS problems and interactive debugging

Evolving a heritage application using Java with RD/z

We wrote this IBM® Redbooks publication for clients who implement the Java™ language support that is provided by CICS® Transaction Server for z/OS® V3.2. Our prime audience is CICS and z/OS system programmers who provide support for Java application development and Java application programmers who need a gentle introduction to Java development for CICS.

In this book, we assume that you have knowledge of z/OS, CICS, UNIX® System Services, and Java.

We start by reviewing the basic concepts of the z/OS, CICS TS V3.2, and Java environments, and introduce new terminology. We then discuss the software and hardware requirements for developing and executing Java applications in CICS TS V3.2. Next we show you how to customize the application development environment, UNIX System Services, MVS™, and CICS.

Additionally, we briefly discuss three possible application development roadmaps: Java application programs that use CICS services, IIOP server applications, and CICS Enterprise Beans.

Subsequent chapters contain an expanded explanation and examples of Java application programs that use CICS services and how to use CICS-supplied Java class library and the Java Virtual Machine (JVM™). We then present a CICS business application that has presentation and business logic.

Finally, we provide guidance on debugging and problem determination.

**INTERNATIONAL
TECHNICAL
SUPPORT
ORGANIZATION**

**BUILDING TECHNICAL
INFORMATION BASED ON
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:
ibm.com/redbooks**